DISEÑO E IMPLEMENTACIÓN DE UN CHATBOT PARA EL SOFTWARE DE IDBOX

(Design and implementation of a chatbot for the IDbox software)

Trabajo de Fin de Máster
para acceder al

MÁSTER EN CIENCIA DE DATOS

Autora: Verónica Pinilla Gómez

Director\es: Steven Van Vaerenbergh
Carlos Alberto Meneses Agudo

Junio - 2020

Resumen

Los chatbots son programas de ordenador que interactúan y establecen una conversación con los usuarios mediante el uso del lenguaje natural. Esta tecnología comenzó en la década los años 60 con el objetivo de mostrar si estos sistemas eran capaces de exhibir un comportamiento inteligente. La evaluación de este comportamiento consistía en ver si dichas máquinas eran capaces de confundir a los usuarios y hacerles creer que estaban hablando con una persona real. A día de hoy, son muchas las empresas que han introducido esta tecnología visto su potencial y beneficios.

En el presente trabajo se ha utilizado una tecnología actual de código abierto para la implementación de un asistente virtual. Para sacar un mayor potencial a la herramienta, ha sido necesario desarrollar conceptos del procesamiento natural del lenguaje y analizar los modelos de redes neuronales que vienen implementados en dicha herramienta. Además, una vez entendidas estas bases, se proporciona el código para que quien desee pueda probarlo y crear su propio chatbot.

Palabras clave: Rasa, chatbot, procesamiento natural del lenguaje, inteligencia artificial.

Abstract

Chatbots are computer programs that interact and establish a conversation with users by using natural language. This technology began in the 1960s with the aim of showing if these systems were capable of exhibiting intelligent behaviour. The evaluation of this behavior was to see if these machines were capable of confusing users and making them believe that they were talking to a real person. Today, many companies have introduced this technology in view of its potential and benefits.

In the present work a current open source technology has been used for the implementation of a virtual assistant. In order to take advantage of the tool's potential, it has been necessary to develop concepts of natural language processing and to analyze the models of neural networks that are implemented in this tool. Furthermore, once these bases are understood, the code is provided so that anyone can try it out and create their own chatbot.

Keywords: Rasa, chatbot, natural language processing, artificial intelligence.

Glosario

Acción Sentencia o contestación que el bot ejecuta en respuesta a las entradas del usuario.

Bolsa de palabras Representación vectorial de un texto que describe cuántas veces aparece cada palabra dentro de un documento.

Chatbot, bot Formado a partir de la unión de *chat* y *robot*. Programa informático que simula una conversación humana.

CLS Token especial que se utiliza para condensar el significado total de una frase.

Credencial Información de autenticación de un usuario para que el asistente esté disponible en plataformas de mensajería.

CRF *Conditional Random Field*. Modelo discriminativo utilizado habitualmente para datos secuenciales.

DIET *Dual Intent Entity Transformer*. Arquitectura multitarea basada en transformadores para la clasificación de intención y reconocimiento de entidades.

Endpoint Extremo final de un canal de comunicación donde una API envía una solicitud y emite la respuesta.

Entidad Término relevante del mensaje del usuario.

Historia Representación de una conversación entre un usuario y el asistente virtual. Los entradas del usuario se expresan con las intenciones correspondientes (y entidades cuando es necesario) mientras que las respuestas del bot se expresan con las acciones correspondientes.

Incrustación de palabras Modelo en el que las palabras se asignan a vectores de números reales.

Intención Propósito del mensaje del usuario.

LSTM *Long Short Term Memory*. Tipo de red neural recurrente capaz de aprender dependencias a largo plazo.

MASK Token especial que se utiliza para ocultar o enmascarar una palabra e intentar predecirla en función del resto.

Modelo de lenguaje Función que asigna una probabilidad a una secuencia de palabras mediante una distribución de probabilidad. Normalmente se utiliza para calcular la probabilidad de la siguiente palabra en una secuencia.

N-gramas Subsecuencia contigua de n elementos de una secuencia de texto.

NLG *Natural Language Generation*. Rama del procesamiento natural del lenguaje que se ocupa de trasformar datos estructurados en lenguaje natural.

NLP *Natural Language Processing*. Campo de la inteligencia artificial que se encarga del procesamiento del texto para que sea legible por las máquinas.

NLU *Natural Language Understanding*. Rama del procesamiento natural del lenguaje que se encarga de la comprensión del lenguaje.

Pipeline Conjunto de técnicas para el procesamiento de datos conectados en serie de manera que la salida de cada método es la entrada de alguna siguiente.

Policy Componente que decide qué acción ejecutar en cada paso de un diálogo.

Señal Identificador en IDbox de una variable.

Slot Clave-valor que se utiliza para almacenar información. Hace referencia a la memoria del bot.

Token Unidad mínima en la que se puede dividir una oración.

Índice general

Resumen iii

Abstract v

1. Introducción 1

2. Estudio del estado del arte 3
- 2.1. Comienzos . . . 4
- 2.2. Aplicaciones . . . 5
- 2.3. Plataformas de desarrollo . . . 5

3. Fundamento teórico 7
- 3.1. Procesamiento Natural del Lenguaje . . . 7
- 3.2. Rasa NLU . . . 8
  - 3.2.1. Elegir la configuración de Rasa NLU . . . 8
  - 3.2.2. Componentes del *pipeline* . . . 9
- 3.3. Rasa CORE . . . 13
  - 3.3.1. Componentes del *policy* . . . 14
- 3.4. Redes neuronales . . . 15
  - 3.4.1. DIET . . . 15
  - 3.4.2. Redes neuronales recurrentes . . . 18

4. Desarrollo 23
- 4.1. IDbox . . . 23
- 4.2. Descripción del alcance del bot . . . 24
  - 4.2.1. Instalación . . . 25
  - 4.2.2. Frases de entrenamiento: `nlu.md` . . . 26
  - 4.2.3. Dominio: `domain.yml` . . . 27
  - 4.2.4. Historias: `stories.md` . . . 28
  - 4.2.5. Acciones personalizadas: `actions.py` . . . 28
  - 4.2.6. Archivos `credentials.yml` y `endpoints.yml` . . . 29
  - 4.2.7. Entrenamiento . . . 29

5. Análisis de los modelos y resultados **31**
5.1. Elección del pipeline 31
5.2. Análisis detallado de los modelos 32
5.3. Resultados 34

6. Conclusiones y trabajo futuro **39**

A. Matrices de confusión **41**
A.1. Entrenamiento 41
A.2. Validación 43

B. Conversación con el bot **45**

Bibliografía **47**

Capítulo 1

Introducción

La popularidad de los chatbots ha aumentado exponencialmente en los últimos años [1], consiguiéndose mantener conversaciones cada vez más indistinguibles de los humanos. De esta manera, las empresas pueden ahorrar mucho tiempo y dinero ya que el objetivo básico de los bots es convertirse en intermediarios ayudando a los usuarios a ser más productivos y resolver tareas de diferentes propósitos.

Las empresas están empezando a utilizar los chatbots como una herramienta de marketing muy poderosa para la captación de clientes. Algunas de las ventajas más notorias que pueden proporcionar son las siguientes:

- **Accesibilidad:** Son fácilmente accesibles. El consumidor puede abrir el sitio web y empezar a hacer preguntas o resolver sus dudas sin tener que realizar acciones tediosas. Por ejemplo, no es necesario seguir instrucciones en las llamadas telefónicas del tipo *"presione la tecla 1, ahora presione la tecla 2..."*.
- **Disponibilidad:** Están disponibles 24 horas al día, los 7 días a la semana. Ejecutan todas las tareas cada vez con la misma eficiencia y rendimiento.
- **Escalabilidad:** Pueden manejar fácilmente miles de consultas al mismo tiempo. El cliente no tiene que esperar a que sea su turno para que le atiendan.
- **Memoria:** Pueden recordar el comportamiento del usuario y proporcionarle respuestas personalizadas. Además, pueden hacer uso de las últimas técnicas de inteligencia artificial y ciencia de datos para desarrollar sistemas de recomendación, entre otros.

Expuestos estos puntos se puede observar lo eficiente que resulta la implementación de chatbots para los consumidores debido a su utilidad y eficiencia que proporcionan. Es por esto que cada vez son más empresas las que están integrando esta tecnología.

La finalidad de este estudio es desarrollar una prueba de concepto de un chatbot para el software de *IDbox* de la empresa *CIC Consulting Informático* que sea capaz de resolver solicitudes básicas pero muy frecuentes relacionadas con la atención del cliente. Se muestra una mejora de la satisfacción de los clientes cuando se les ofrece la posibilidad de acceder a un chat donde pueden ser guiados y resolver sus dudas sin tener que ponerse en contacto con un empleado y esperar a que este les conteste [2].

Para llevar a cabo esto, ha sido fundamental iniciarse en el campo del procesamiento del lenguaje natural, entendiendo todas las técnicas necesarias que se necesitan para interpretar el lenguaje humano por las máquinas.

La memoria está dividida en cinco apartados. Se comienza el trabajo (Capítulo 2) exponiendo las principales aplicaciones de los asistentes virtuales, así como algunos ejemplos conocidos y plataformas para implementarlos. En el Capítulo 3 se encuentra el desarrollo teórico necesario para comprender las etapas para procesar el lenguaje natural y se describen y se desarrollan los modelos de inteligencia artificial implicados en la creación del asistente. Seguido en el Capítulo 4, se describe el chatbot realizado para la empresa en la que he realizado las prácticas, exponiendo el alcance del mismo y los archivos y formatos necesarios para su creación. Finalmente se exponen los resultados, conclusiones y futuras mejoras en los Capítulos 5 y 6.

Capítulo 2

Estudio del estado del arte

Un chatbot es un programa informático, especialmente utilizado como apoyo a páginas web, basado en inteligencia artificial que simula una conversación humana [3]. En [4] se proporciona un marco básico que describe las tres funciones principales que se esperan que los chatbots deben llevar a cabo.

- **Agente de Diálogo:** Debe captar la petición del mensaje del usuario. Los bots reciben una entrada textual u oral, que debe ser analizada con herramientas del procesamiento del lenguaje natural para generar las respuestas adecuadas.

- **Agente Racional:** El asistente debe tener acceso a una base de datos externa de manera que pueda responder adecuadamente a las preguntas de los usuarios. Además, debe ser capaz de almacenar información específica del contexto del mensaje, por ejemplo, el nombre del usuario, fechas, ciudades, etc., para dar la respuesta adecuada.

- **Agente Personificado:** El lenguaje del bot debe ser lo más natural y real posible. Tiene que parecerse lo máximo a una conversación humana por lo que es necesario añadirle personalidad. Hoy en día, los desarrolladores se centran en crear una confianza con los usuarios y dar la impresión de que están hablando con una persona. Por ejemplo, incluso a los primeros bots, los cuales se debaten a continuación, recibieron nombres (Eliza, Parry, ALICE, etc.) para satisfacer esta condición.

A continuación, se realiza un repaso de la historia de los bots, sus aplicaciones y plataformas de desarrollo más conocidas mediante la información extraída del libro [5].

2.1. Comienzos

Aunque los chatbots han ganado popularidad recientemente y parece que su aparición sea bastante actual, se lleva trabajando décadas con esta tecnología. El primero que empezó a plantear si las máquinas eran capaces de pensar fue Alan Turing en 1950 en el artículo *Computing machinery and intelligence* [6]. En dicho artículo, se describe el juego llamado *The Imitation Game*. Se juega con tres personas, un hombre, una mujer, y un interrogador. El objetivo del interrogador es determinar cuál de ellos es la mujer y cuál el hombre. Posteriormente, se cuestiona qué ocurre si se sustituye al hombre por una máquina. Turing afirmaba que si la máquina conseguía hacer creer al interrogador que estaba hablando con una persona en vez de con una computadora entonces se podía considerar que la máquina piensa. Esta prueba, recibe el nombre de *Test de Turing* que trata de exhibir un comportamiento inteligente equivalente o indistinguible al de un humano.

En 1966 se creó el primer chatbot llamado Eliza por Joseph Weinzebaum. Fue programada para entender los patrones de la consulta del usuario. Utilizando una metodología de sustitución y un algoritmo de emparejamiento de patrones, este elegía la respuesta. Por ejemplo, si el usuario decía *"Mi madre cocina muy bien"*, Eliza recogía la palabra *madre* y respondía con una pregunta abierta del tipo *"Cuéntame más sobre tu familia"*, dando a los usuarios una falsa ilusión de comprensión por parte del bot, aunque el proceso era mecanizado. Años más tarde, en 1972, Kenneth Colby se basó en la idea de Eliza para crear un programa de lenguaje natural llamado Parry que simulaba el pensamiento de un individuo con esquizofrenia.

En 1990 se instituyó el Premio Loebner de Inteligencia Artificial para destacar trabajos que alcanzasen la primera instancia de un test de Turing ofreciendo un premio de 100.000 dólares. Durante los primeros años el ganador fue Joseph Weinzebaum, el creador del chatbot Eliza.

Pocos años después, en 1995, el creador también galardonado con el Premio Loebner e inspirado en Eliza Richard Wallace, creó A.L.I.C.E (*Artificial Linguistic Internet Computer Entity*). Al contrario que Eliza, este utilizaba procesamiento natural del lenguaje generando así respuestas más complejas y sofisticadas. Además, fue revolucionario por ser de código abierto.

En 2001, Smarterchild, un robot inteligente desarrollado por ActiveBuddy, se distribuyó ampliamente a través de las redes de mensajería y SMS. La implementación original creció rápidamente para incluir respuestas a preguntas sobre noticias, el tiempo, horarios de películas, datos deportivos detallados, así como proporcionar acceso instantáneo a una variedad de herramientas (calculadora, traductor, etc.).

Actualmente, las grandes empresas como Amazon (Alexa), Google (Google Assistant), Microsoft (Cortana) o Apple (Siri) han desarrollado sus propios asistentes.

2.2. Aplicaciones

Aunque principalmente se encuentre el uso de los chatbots a nivel empresarial, es un servicio que cada vez está cobrando más fuerza en todos los sectores [2]. Su disponibilidad, usabilidad y sencilla incorporación en los principales canales de mensajería instantánea han facilitado que diversos sectores se beneficien de las ventajas que proporcionan los asistentes virtuales.

El sector de servicios de atención al cliente es sin duda donde ha irrumpido con más fuerza. Los chatbots son capaces de resolver los problemas o cuestiones de los usuarios de forma inmediata, pudiendo manejar miles de solicitudes simultáneamente con respuestas similares. Esto hace que sean idóneos para tratar con preguntas frecuentes.

Otra aplicación es la del comercio online. Las grandes marcas están optando por esta opción ya que permiten al usuario encontrar un producto sin necesidad de hacer una búsqueda completa en la web. La dinámica que sigue el bot es realizar preguntas al usuario, por ejemplo en el sector textil, qué tipo de prenda desea, color, talla, etc. y este le ofrece al usuario artículos que cumplen dichas características.

Un tercer campo donde están cogiendo notoriedad es el de la salud. Los bots mediante la realización de preguntas al paciente analizan los síntomas para agilizar el diagnóstico. No sustituyen a los médicos, pero gracias a estos asistentes se agiliza considerablemente la atención al paciente.

Finalmente, el último sector que se expone de ejemplo es el de la educación. Los estudiantes pueden resolver sus dudas en cualquier momento del día facilitándoles además fuentes externas con información adicional de refuerzo. De esta manera se puede ver en qué materias necesitan más ayuda los alumnos. Permiten también facilitar los procesos de inscripción y admisión que normalmente son muy engorrosos resolviendo a los estudiantes las dudas que tengas acerca de los trámites y ayudarles en el proceso.

2.3. Plataformas de desarrollo

Expuestos los beneficios de los chatbots, se muestran algunos de los marcos más conocidos para diseñar y desarrollar un bot.

- **Dialogflow**[1]**:** Creada en 2010, se trata de una plataforma tanto para gente especializada como para entidades que necesitan implementar un bot rápidamente sin tener mucho conocimiento previo. Se trata de la herramienta creada por Google por lo que utilizan su amplia experiencia en el mundo de la inteligencia artificial y el gran volumen de datos que disponen. Dialogflow es fácil de usar y soporta más de 20 idiomas, además del procesado tanto de texto como de voz. Por todo

[1] https://dialogflow.com/

esto hace que sea una de las herramientas disponibles más potentes en el mercado en este momento. Su precio varía en función del número de solicitudes y la duración total del audio procesado.

- **IBM Watson**[2]**:** Esta tecnología fue lanzada el mismo año que Dialogflow, en el año 2010. Está construida en base a una red neural utilizando para su entrenamiento un billon de palabras de la Wikipedia. Se puede desplegar en una página web, en una aplicación móvil, en el teléfono, en canales de mensajería y en herramientas de servicio al cliente. Soporta 13 idiomas. Proporciona un SDK (*Software Development Kit*) para los desarrolladores que se puede utilizar en Java, Python e iOS. En este caso se ofrecen planes de diferentes precios ajustados a las necesitades de cada empresa.

- **Botkit**[3]**:** Desarrollada en el año 2015, es una de las herramientas líderes de entorno abierto recientemente adquirida por Microsoft. BotKit es un SDK basado en NodeJs que soporta la implementación en canales como Slack, Facebook Messenger, Telegram y otras plataformas de mensajería. Botkit también proporciona un plugin de chat web que puede incrustarse en cualquier sitio web.

- **RASA Stack**[4]**:** Rasa es un marco de trabajo de código abierto implementado en el año 2018. Tiene dos componentes principales: Rasa NLU y Rasa Core. Rasa NLU es responsable de la comprensión del lenguaje natural. Rasa Core se encarga en cambio del manejo del diálogo. Dichas componentes se pueden utilizar por separado. Una de las mayores ventajas de utilizar Rasa es que el chatbot puede ser desplegado en el propio servidor manteniendo todos los componentes en local. Además, está escrito en Python.

Después de haber analizado diferentes marcos para el desarrollo del bot, se decidió implementarlo utilizando Rasa fundamentalmente por las siguientes razones:

1. Rasa no es de pago, lo cual a larga vista para la empresa es muy beneficioso.

2. Permite customizar todo el código a medida. Se pueden añadir funciones personalizadas, cambiar los modelos o su entrenamiento.

3. Está escrito en Python, lenguaje que manejamos tanto el equipo de la empresa con el que he trabajado como yo.

[2]https://www.ibm.com/watson/how-to-build-a-chatbot
[3]https://botkit.ai/
[4]https://rasa.com/

Capítulo 3

Fundamento teórico

Rasa se trata de una librería de código abierto por lo que permite customizar todo su código. Se pueden añadir funciones personalizadas, implementar modelos propios, cambiar los parámetro de los modelos ya existentes, los pasos a realizar para la extracción de datos, etc.

Estas fueron las principales razones por las que se decidió implementar el bot utilizando esta librería, no solo por la parte de la personalización, sino también por la necesidad de entender cada una de las etapas que conforman el procesamiento del lenguaje natural para poder crear en un futuro implementaciones más complejas.

En los siguientes apartados se describirán todas las fases que deben de realizarse tanto para el apartado de la comprensión como en el de la generación de la respuesta dentro de la herramienta de Rasa, así como los modelos de inteligencia artificial que se implementan.

Para el desarrollo de este capítulo se ha utilizado la documentación disponible en el sitio web oficial de Rasa [7] y en su repositorio *Github* [8].

3.1. Procesamiento Natural del Lenguaje

El Procesamiento Natural del Lenguaje (*Natural Language Processing*, abreviado como NLP) [9, 10] es una parte fundamental para la construcción de los chatbots. Es un campo de la inteligencia artificial que se encarga de las interacciones entre las máquinas y los humanos. El NLP se utiliza para el reconocimiento de voz o texto permitiendo así que una computadora sea capaz de analizar y procesar el lenguaje humano.

El procesamiento del lenguaje está considerado un problema difícil de la informática. El lenguaje natural raramente es preciso o sigue unas reglas. No solo basta con entender las palabras, sino que hace falta entender también el contexto. Si bien los seres humanos pueden aprender fácilmente un idioma, la ambigüedad del lenguaje y el no tener reglas precisas para todos los casos son lo que hacen que el NLP sea difícil de aplicar para las máquinas.

El procesamiento del lenguaje se considera el cerebro de los chatbots ya que procesan los datos en bruto y después de una serie de tratamientos y etapas se pueden desarrollar modelos sobre estos datos.

El NLP está dividido principalmente en dos subgrupos que son la Comprensión del Lenguaje Natural (*Natural Language Understanding*) y la Generación del Lenguage Natural (*Natural Languaje Generation*). El primero de ellos, el NLU, se encarga de comprender el significado que hay detrás del texto escrito o hablado. Una máquina o modelo no puede tomar los datos en bruto, es necesario aplicar un pre-procesamiento para que la máquina pueda interpretarlo fácilmente. La herramienta que se encarga de realizar este trabajo es Rasa NLU. Por el otro lado, el NLG transforma los datos estructurados en lenguaje natural. Para realizar esta labor, en este caso se utiliza Rasa Core.

3.2. Rasa NLU

Rasa NLU es la herramienta que permite realizar la clasificación de intenciones y extracción de entidades. Esta librería permite preparar los datos de entrenamiento para el chatbot. Mediante el apartado de pipeline del archivo `config.yml` se eligen los pasos a realizar en el procesamiento del texto y se selecciona el modelo a entrenar. Este archivo es uno de los diferentes ficheros que componen un proyecto de Rasa.

En los siguientes apartados se examinan qué dos tipos de configuraciones hay principalmente y se detalla paso a paso la configuración seleccionada para el chatbot desarrollado.

3.2.1. Elegir la configuración de Rasa NLU

Un *pipeline* define las diferentes componentes que procesan el mensaje del usuario de manera secuencial, conduciendo finalmente a la clasificación del mensaje en términos de intención y a la extracción de entidades. Las intenciones describen cómo deben clasificarse los mensajes del usuario y las entidades son conceptos del texto que se quieren destacar, como pueden ser nombres propios, fechas, colores, etc.

Existen principalmente dos tipos de pipelines, los que utilizan incrustaciones de palabras pre-entrenadas (*Pre-trained Embeddings*) y los que no (*Supervised Embeddings*). Fundamentalmente se diferencian en el enfoque que toman en cuanto a la extracción de las características para entrenar los modelos de predicción de la intención y extracción de entidades.

- **Pre-trained Embeddings**

  Este tipo de configuración utiliza librerías como la de *spaCy* para cargar modelos de lenguaje pre-entrenados que luego se utilizan para representar cada palabra del mensaje del usuario como incrustación de

palabras, más conocido en inglés como *word embedding*. Las incrustaciones de palabras son representaciones vectoriales de las palabras, esto es, cada palabra se transforma en un vector numérico. Además, las incrustaciones captan los aspectos semánticos y sintácticos de las palabras, es por esto que palabras similares son representadas por vectores similares [11].

Las incrustaciones de palabras son específicas para cada idioma, es decir, dependen del idioma con el que fueron entrenadas.

Al utilizar incrustaciones de palabras pre-entrenadas, uno se puede beneficiar de los recientes avances. Además, dado que las incrustaciones están ya entrenadas, el modelo para la clasificación de la intención requiere menor entrenamiento también. Si se tienen pocos datos de entrenamiento, lo cual es muy habitual al inicio del desarrollo, se pueden llegar a obtener aun así resultados robustos.

Lamentablemente, no se dispone de incrustaciones pre-entrenadas buenas para todos los idiomas, ya que fundamentalmente están entrenadas sobre conjuntos de datos públicos y principalmente la mayoría se encuentran en inglés. Las incrustaciones tampoco cubren palabras de un dominio especifico como pueden ser nombres de productos o acrónimos. En este caso es mejor entrenar incrustaciones de palabras propias basadas en el corpus de entrenamiento sin utilizar modelos pre-entrenados.

- **Supervised Embeddings**

  Estos pipelines, en lugar de utilizar incrustaciones pre-entrenadas y entrenar un modelo de clasificación en base a estas, crea las incrustaciones desde cero apoyándose en el conjunto de entrenamiento disponible.

  A diferencia de la anterior configuración, esta soporta mensajes con múltiples intenciones. Por ejemplo, *Hola, ¿qué tiempo hace hoy?* tiene las intenciones de *saludo* y *preguntar_tiempo*.

  Tal y como se ha mencionado, como estas configuraciones crean las incrustaciones de palabras desde cero, se necesitan muchos más datos de entrenamiento que en el caso de las incrustaciones pre-entrenadas. Sin embargo, como tan solo utiliza el corpus de entrenamiento para crear las incrustaciones se adapta mucho mejor a mensajes específicos del propio dominio. Además, es independiente del idioma, por lo que un escaso desarrollo de incrustaciones pre-entrenadas para el idioma deseado no es un problema para este tipo de pipelines.

3.2.2. Componentes del *pipeline*

Las componentes utilizadas para el desarrollo del bot han sido las siguientes:

```
language: es
pipeline:
- name: WhitespaceTokenizer
- name: RegexFeaturizer
- name: LexicalSyntacticFeaturizer
- name: CountVectorsFeaturizer
- name: CountVectorsFeaturizer
  analyzer: "char_wb"
  min_ngram: 1
  max_ngram: 4
- name: DIETClassifier
  epochs: 100
  use_masked_language_model: False
- name: EntitySynonymMapper
```

Ninguna de estas componentes pertenece a un modelo de lenguaje pre-entrenado, por lo que se ha escogido una configuración supervisada. Las razones de por qué se ha elegido este tipo de estructura se presentan en el apartado de resultados, detallado en el Capítulo 5.

Aunque para esta configuración supervisada el parámetro del idioma no afecta nada, Rasa recomienda aun así declararlo.

En base a en esta estructura se proceden a definir y explicar las partes que intervienen en esta primera fase del procesamiento del texto.

Tokenización: `WhitespaceTokenizer`

La tokenización, también conocida como segmentación, es uno de los conceptos más simples pero básicos del NLP. Consiste en dividir la cadena de caracteres en unidades más pequeñas llamadas *tokens*, con las que se trabajarán posteriormente. Fundamentalmente, los tokens son las palabras de la oración. Para el español es una tarea relativamente sencilla ya que se puede principalmente separar las palabras por espacios y caracteres que no pertenecen al alfabeto. En el caso del inglés, por ejemplo, se convierte en una tarea más compleja. Si se tiene la expresión *don't* el algoritmo tiene que identificar las locuciones *do* y *n't*.

Ejemplo

```
text = 'La tokenización (segmentación) es uno de los
   conceptos básicos del N.L.P.'
tokens = extract_tokens(text)

print(tokens)
>>> ['la', 'tokenización', 'segmentación', 'es', 'uno',
   'de', 'los', 'conceptos', 'básicos', 'del', 'n.l.p']
```

La función `extract_tokens` en primer lugar, pasa todo el texto a minúsculas mediante el atributo `.lower()`. En segundo lugar, utiliza la función `re.sub()` de la librería `re` *(Regular expression operations)*. Con esta función, mediante el uso de expresiones regulares se decide qué es y qué no es un token. Si se identifica una cadena de caracteres con las expresiones regulares incluidas se reemplaza por un espacio en blanco. De esta manera se llama entonces al atributo `.split()` para dividir la cadena resultante en tokens.

Se observa cómo se ha introducido a propósito las siglas *N.L.P* y lo ha detectado correctamente como un único token.

Expresiones regulares: `RegexFeaturizer`

Es necesario que el texto haya sido previamente tokenizado para aplicar esta componente. Este elemento crea una lista sobre las expresiones regulares definidas en el conjunto de entrenamiento. Estas expresiones regulares se declaran en el archivo `nlu.md` tal y como se ve en el Capítulo 4 donde se detalla el desarrollo del código.

Para cada expresión regular y cada token, se señala con un 1 si el token coincide con la expresión regular y con un 0 en caso contrario. Se utiliza como entrada opcional en la componente referente a la clasificación de la intención y extracción de las entidades (`DIETClassifier`). Estas características extraídas reciben el nombre de características dispersas (*sparse features* en inglés), ya que la mayoría de los valores son ceros.

```
function RegexFeat(set_patters, set_tokens, vec)
    for each pattern in set_patters do
        for each token in set_tokens do
            vec[token][pattern] = 1.0
```

Ejemplo

```
patterns = ["[a-zA-Z]{4}[0-9]{4}",
            "[0-9]{4}-[0-9]{2}-[0-9]{2}"]
tokens = [cic0005, entre, 2017-05-05, y, 2018-05-05]
vec = np.zeros([len(tokens), len(patterns)])

RegexFeat(patters, tokens, vec)
>>> [[1.0, 0.0],
     [0.0, 0.0],
     [0.0, 1.0],
     [0.0, 0.0],
     [0.0, 1.0]]
```

Clasificador de intención: `LexicalSyntacticFeaturizer`

Se requiere de nuevo que el texto esté previamente tokenizado. Este elemento crea características para la extracción de entidades. Se mueve bajo el protocolo de ventana deslizante sobre cada token del mensaje del usuario y crea características. Algunas características son por ejemplo los primeros y últimos caracteres del token. Además, ya que el extractor de características se mueve sobre los tokens del mensaje mediante una ventana deslizante, se pueden definir características para los datos de la ventana como por ejemplo, para el token anterior, actual y siguiente. Las características creadas en este caso también son dispersas.

Clasificador de intención: `CountVectorsFeaturizer`

Esta configuración utiliza dos instancias de `CountVectorsFeaturizer`. La primera de ellas extrae características del texto basándose en las palabras. La segunda en cambio, se basa en caracteres n-gramas. En principio el segundo extractor es más poderoso, pero decidieron mantener el primer extractor también para hacerlo más robusto.

Para utilizar esta componente se necesita nuevamente que le texto esté previamente tokenizado. Esta componente crea un modelo de "bolsa de palabras" utilizando la función `CountVectorizer()` de la librería *Scikit-learn* [12]. De nuevo, las características creadas por esta componente son dispersas.

Clasificador de intención: `DIETClassifier`

DIET (*Dual Intent Entity Transformer*) [13, 14] es un modelo que sirve tanto para la extracción de entidades como para la clasificación de la intención. Se trata de un modelo basado en transformadores. Para el caso de la extracción de las entidades se emplean además capas CRF, *Conditional Random Fields* [15]. Como se trata de un modelo complejo implementado por el propio Rasa se analizan sus características en la sección 3.4.1.

Sinónimos: `EntitySynonymMapper`

Esta componente modifica las entidades extraídas mediante la componente anterior por los sinónimos definidos en el archivo `nlu.md`. De esta manera, se asegura que los valores de las entidades detectados se asignen a un mismo valor siempre. Este proceso simplifica las acciones posteriores a realizar en la sección del NLG, es decir, en la generación de la respuesta del bot. El chatbot a desarrollar debe ser capaz de entender variaciones en las entidades.

```
function EntitySynonym(dic, set_synonyms)
    for entity in dic[entities] do
        entity_value = entity["value"]
        if entity_value in synonyms do
            entity["value"] = synonyms[entity_value]
```

Ejemplo

```
synonyms = {"datos temporales": "serie",
  "cifras temporales": "serie"}

EntitySynonym(dic1, synonyms)
>>> [{"text": "Quiero la gráfica de la serie temporal",
    "intent": "grafica_serie",
    "entities": [{"entity": "tipo_grafica",
                  "start": 24,
                  "end": 38,
                  "extractor": "DIETClassifier",
                  "value": "serie"}]

EntitySynonym(dic2, synonyms)
>>> {"text": "Necesito los datos temporales de un id",
    "intent": "grafica_serie",
    "entities": [{"entity": "tipo_grafica",
                  "start": 13,
                  "end": 29,
                  "extractor": "DIETClassifier",
                  "value": "serie"}]
```

Tanto `dic1` como `dic2` son diccionarios con la misma estructura que la salidas que se observan pero el valor de la entidad para cada caso es el original de cada frase, *serie temporal* y *datos temporales* respectivamente.

3.3. Rasa CORE

El manejo del diálogo y cómo elige el bot la respuesta a dar al usuario se realiza mediante la herramienta llamada Rasa Core. A través de esta, se construye un modelo que decide entre un conjunto de acciones (respuestas) cuál es la más adecuada basándose en las anteriores entradas del usuario. Las componentes que se desean utilizar también hay que especificarlas en el archivo `config.yml`, pero en este caso hay que indicarlas en el apartado de *policies*, el cual define las diferentes componentes que se utilizan para el manejo del diálogo.

3.3.1. Componentes del *policy*

La configuración elegida para el manejo del diálogo ha sido la siguiente:

```
policies:
- name: MemoizationPolicy
  max_history: 3
- name: KerasPolicy
- name: MappingPolicy
- name: FallbackPolicy
  nlu_threshold: 0.5
  core_threshold: 0.5
  fallback_action_name: "my_fallback_action"
```

A continuación, se explican cada una de las componentes.

MemoizationPolicy

La componente `MemoizationPolicy` [sic] es una de las más simples de las que se dispone. Imita las historias con las que fue entrenado el modelo. Para ello, hay que establecer primero el parámetro `max_history`. Este parámetro controla cuánta historia del diálogo mira el modelo para decidir qué acción tomar.

Dependiendo del número fijado para dicho parámetro, busca que coincida el fragmento de la historia actual con las historias proporcionadas en el archivo de datos de entrenamiento. Si hay algún fragmento que coincide, predice la próxima acción de la historia con una confianza de 1.0, de lo contrario no predice ninguna y devuelve una confianza de 0.0.

MappingPolicy

Se utiliza para asociar directamente las intenciones a una determinada acción. Por ejemplo, si el usuario saluda, el bot tiene que responder con un saludo también.

Esta componente también es responsable de ejecutar las acciones predefinidas `action_back` y `action_restart`, que son las que se encargan de deshacer el último mensaje del usuario y reiniciar la conversación respectivamente.

FallbackPolicy

Esta componente es muy importante para la gestión del diálogo. Cuando se trabaja con chatbots es difícil evitar la situación en la que los usuarios piden algo para lo que el asistente no ha sido diseñado realmente o en la que los usuarios introducen el texto de tal manera que hace que el entendimiento o la gestión de diálogo se vean confundidos. Si bien es complejo crear un asistente que sea capaz de manejar todas las situaciones posibles,

es importante que al menos el bot reconozca dicha situación y sea capaz de solventarla favorablemente.

De esto se encarga esta componente. Permite fijar un umbral para los modelos tanto de entendimiento como de manejo del diálogo. Si la predicción no sobrepasa dicho umbral, se devuelve una acción *fallback* predefinida. Esta acción puede ser una respuesta tipo: *"Lo siento, no te he entendido. ¿Puedes reformular la pregunta?"*.

KerasPolicy

Utiliza una red neuronal implementada mediante la librería *Keras* para predecir la siguiente acción. Para realizar la predicción se tienen en cuenta diferentes propiedades como por ejemplo:

- Cuál fue la última acción.
- Qué intención y entidades fueron predichas para la entrada actual del usuario.
- Qué *slots* están establecidos en este momento. Los slots se pueden definir como la memoria del bot, no obstante se tratan con mayor detalle en el Capítulo 4 donde se detalla el desarrollo.

La arquitectura por defecto del modelo se basa en celdas LSTM (*Long Short Term Memory*) [16]. En el siguiente apartado se explica la estructura y funcionamiento de dichas celdas.

3.4. Redes neuronales

Tanto para la clasificación de la intención como para la respuesta que debe devolver el bot al usuario se utilizan modelos de redes neuronales. Aunque el concepto de red neuronal sea el mismo para los dos modelos, la estructura y el tipo de red es totalmente distinta en ambos casos.

Por un lado, para la clasificación de la intención y extracción de las entidades se utiliza una estructura basada en transformadores llamada DIET tal y como se ha mencionado previamente. Por el otro lado, para el manejo del diálogo se utilizan celdas LSTM que son un tipo de redes neuronales recurrentes.

A continuación, se desarrollan ambas estructuras.

3.4.1. DIET

Es una arquitectura multitarea basada en transformadores que permite realizar tanto la clasificación de la intención como la extracción de entidades simultáneamente. Está compuesta de diferentes componentes, las cuales se pueden modificar, siendo así una arquitectura muy flexible.

La mayoría de modelos de lenguaje requieren una gran potencia de cálculo, ya que fundamentalmente están implementados en base a redes neuronales recurrentes. DIET es diferente por los siguientes aspectos:

- Es una arquitectura modular que permite a los desarrolladores tener más flexibilidad.
- Se obtienen resultados semejantes en precisión y rendimiento que modelos de lenguaje pre-entrenados de gran escala.
- Se entrena hasta 6 veces más rápido que dichos modelos de lenguaje.

DIET utiliza un modelo de secuencia que tiene en cuenta el orden de las palabras, ofreciendo así un mejor rendimiento. También es un modelo más compacto con una arquitectura modular "enchufar, conectar y usar" (*plug-and-play*). Por ejemplo, se puede usar DIET para hacer tanto la clasificación de intenciones como la extracción de entidades o puede configurarse para realizar solo una de estas tareas.

La arquitectura de DIET se puede visualizar en la Figura 3.1. Se puede observar como la función de pérdida total a minimizar está compuesta por tres componentes que son: la pérdida de la *entidad*, *intención* y *máscara*.

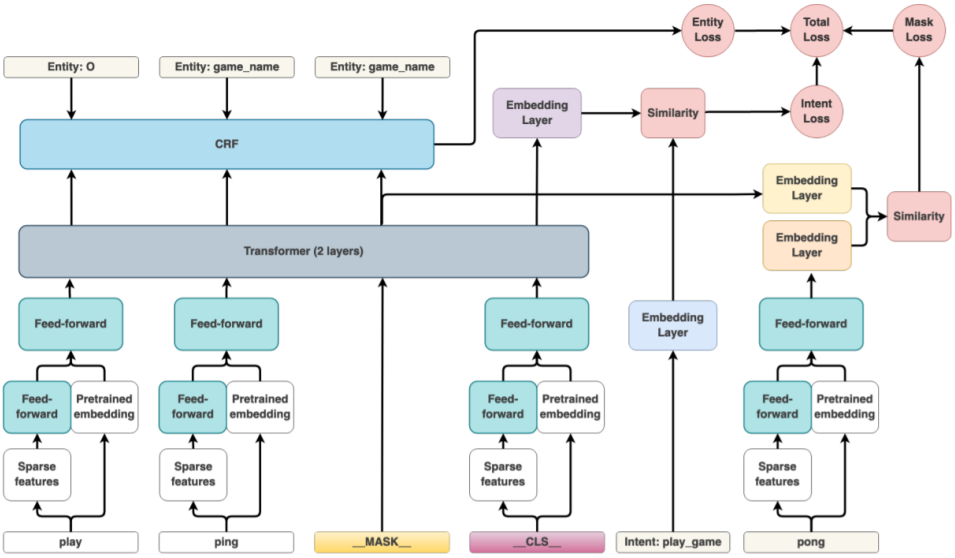

Figura 3.1: Diagrama del modelo de DIET. Encontrado en [14].

Pérdida de la entidad

Hay tres capas intermedias que vinculan las frases de entrada con la función de pérdida de la entidad que son:

1. **Camino del token individual**
   Esta capa está dividida a su vez en dos bloques. El primero de ellos es una red neuronal pre-entrenada que puede ser BERT [17], GloVe [18] o ConverRT [19]. Todos estos algoritmos reciben un token y dan como salida un vector numérico. En el segundo bloque se utilizan las características dispersas extraídas mediante la componente `CountVectorsFeaturizer` y se pasan por una red pre-alimentada, *feed-forward*[1] en inglés. Finalmente, estos dos bloques se concatenan y se introducen a una nueva red feedforward.

2. **Capa transformadora**
   Las salidas de cada token pasan a ser las entrada de una red transformadora. Los transformadores están basados en el concepto de la atención. La atención es un mecanismo que determina en que partes de la entrada de texto hay que enfocarse más para producir la salida correcta, es decir, como de relevante es cada palabra en función del resto de palabras de la frase. Para más información sobre transformadores y el concepto de la atención puede consultarse el artículo [20].

3. **Campo condicional aleatorio**
   Mediante el paquete de *Addons* de la librería *Tensorflow* se implementa un CRF para la clasificación de la entidad utilizando las salidas de la red transformadora. La función de pérdida se obtiene combinando los valores reales de las entidades y los obtenidos por el modelo de CRF.

Pérdida de la intención

Para este apartado se utiliza un token de clase especial, denotado como `__CLS__`. La idea detrás de este token especial es resumir toda la oración para así obtener una representación numérica que sustituye toda la frase de entrada. Este token especial sigue el mismo camino que los tokens individuales, no obstante, la salida de las incrustaciones pre-entrenadas y las características dispersas son ligeramente diferentes

- La salida de la incrustación pre-entrenadas depende del modelo pre-entrenado elegido. Por ejemplo, en el caso de GloVe se toma la media de todas las incrustaciones de palabras.

- Las características dispersas del token de clase especial son la suma de todas las características dispersas de los tokens individuales. Estas características, al igual que ocurre en el anterior apartado, pasan por una red feedforward.

Dado que el token especial es un resumen de toda la frase de entrada, a partir de esta se tiene que poder predecir la intención.

[1] Red neuronal artificial donde las conexiones entre las neuronas no forman un ciclo.

De nuevo estos dos bloques pasan por una red feedforward, luego por la capa transformadora y finalmente por una capa de incrustación que determina la intención predicha. Al mismo tiempo, la intención real va a través de otra capa de incrustación, de esta manera se puede calcular la similitud de ambas incrustaciones y por tanto, calcular la función de pérdida entre las salidas de las dos incrustaciones.

Pérdida de la máscara

El uso de este token, denotado como `__MASK__`, sirve para que el modelo también pueda ser entrenado como modelo de lenguaje. Un modelo de lenguaje es un modelo que se utiliza para predecir el próximo token más adecuado para una oración dados los tokens anteriores. Durante el entrenamiento, el modelo "enmascara" aleatoriamente algunas palabras y el objetivo del algoritmo es predecir cuál es la palabra original que se está enmascarando.

El token de la máscara atraviesa la red transformadora y luego una capa de incrustación. A su vez, el token aleatoriamente enmascarado transcurre, como en el resto de los tokens de los apartados anteriores, por el camino ya detallado acabando también en una capa de incrustación. Al igual que antes, se puede calcular la similitud y una función de pérdida entre ambas incrustaciones. Cuanto menor sea este error, mejor será el modelo para predecir el token enmascarado.

Para finalizar, dos apuntes importantes sobre las redes feedforward que se utilizan a lo largo del modelo. En primer lugar, las redes tienen un *dropout* del 80 % de las conexiones, lo que hace que sean más ligeras. Este porcentaje viene establecido por defecto pero puede ser cambiado indicándolo en en la configuración del modelo en el pipeline. En segundo lugar, las dos redes feedforward que se aplican para cada token, al tratarse de dos diferentes no comparten pesos, pero sí se comparten dichos pesos para todos los tokens.

3.4.2. Redes neuronales recurrentes

Las redes neuronales recurrentes [21], a diferencia de las redes feedforward, pueden trabajar con entradas de longitud variable y permiten conexiones arbitrarias entre las neuronas, pudiéndose incluso crear ciclos. Esta última propiedad implica que existen retroalimentaciones o lo que esto significa, conexiones con capas ya vistas anteriormente. Esto es práctico para los casos en los que la salida de una capa puede ser útil para alguna otra capa que se encuentra en algún instante posterior.

En una conversación o secuencia de texto, los datos están correlacionados, es decir, la siguiente palabra depende de la palabra anterior. Las redes neuronales recurrentes poseen un cierto tipo de memoria y es por esto que son utilizadas para lidiar con este tipo de problemas.

Otra diferencia respecto a las arquitecturas de las redes feedforward es que estas están diseñadas para que los datos de entrada y de salida tengan siempre el mismo tamaño. No obstante, un diálogo se caracteriza por ser un tipo de dato de longitud variable. Es por todo esto que es muy popular el uso de las redes recurrentes en el campo del NLP.

Para realizar esta labor, las redes neuronales recurrentes utilizan el concepto de recurrencia para generar la salida, también conocida como activación. La red no solo utiliza la entrada actual, sino que emplea también la activación generada en la iteración previa.

Antes de nada, hay que definir que es un instante de tiempo t. Este parámetro t se trata de un número entero que define la posición de un elemento dentro de una secuencia. Se define entonces x_t e y_t como la entrada y predicción respectivamente de la red recurrente en el instante de tiempo t. La salida a_t se define entonces como

$$a_t = f(W_{aa} \cdot a_{t-1} + W_{ax} \cdot x_t + b_a)$$

donde f es la función de activación y W_{aa}, W_{ax} y b_a son los coeficientes de la red.

En la parte derecha de la Figura 3.2 se observan varios bloques. El propósito de mostrar los diversos bloques es visualizar mejor qué entradas y qué salidas intervienen en los diferentes instantes de tiempo, aunque la estructura y coeficientes de la red es la misma para cada bloque.

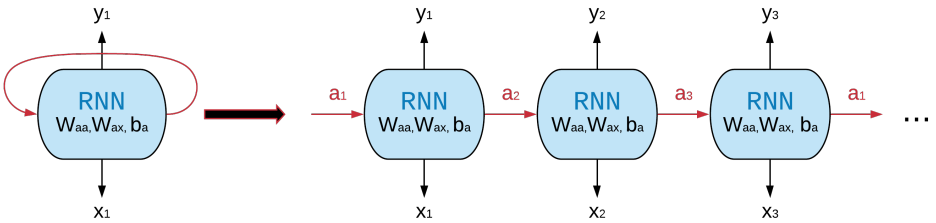

Figura 3.2: A la izquierda, la notación abreviada utilizada frecuentemente para las redes neuronales recurrentes. A la derecha, la notación desplegada.

Se observa que en cada instante de tiempo t la red tiene dos entradas, por un lado el dato actual x_t, y por el otro lado la activación anterior a_{t-1}, indicada con las flechas de color rojo. Este último elemento corresponde a la memoria de la red.

Como bien se ha mencionado, la estructura de la red neuronal es una sola, por lo que los coeficientes a calcular son los mismos para cada instante de tiempo. Es por esto que en lugar de visualizar la red en los diferentes instantes de tiempo, se utiliza la representación de la izquierda de la Figura 3.2, en donde la flecha roja indica la dependencia con la activación generada en un instante anterior.

LSTM

El principal problema de las redes recurrentes básicas es que solo disponen de una memoria a corto plazo. Hay en ocasiones que no es suficiente con tener únicamente información justo de los pasos anteriores y se necesita un mayor contexto. Para solventar este inconveniente, se introdujeron las estructuras LSTM propuestas por Hochreiter y Schmidhuber en 1997 [22]. Estas son capaces de recordar un dato relevante en la secuencia y preservarlo por varios instantes de tiempo. Esto es, disponen tanto de memoria a corto plazo como de largo. Las LSTMs también tienen la estructura de cadena vista en las redes recurrentes, pero a diferencia de estas, en lugar de tener una sola capa interna, estas poseen cuatro diferentes. Se analizan en profundidad las diferentes capas a continuación.

Mediante la Figura 3.3 se visualiza la arquitectura básica de una celda LSTM.

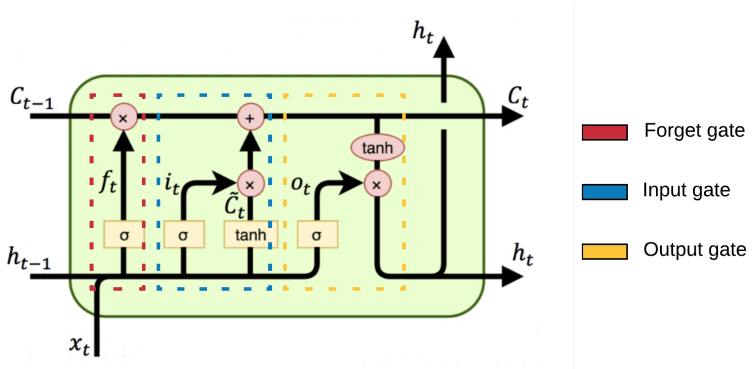

Figura 3.3: Estructura de una celda LSTM. Encontrado en [16].

La clave de las celdas LSTM es la línea horizontal superior llamada *celda de estado*. Esta se puede entender como una banda transportadora a la que se le pueden tanto añadir como quitar datos asociados a la memoria de la red. Para añadir o eliminar datos de esta memoria se utilizan varias compuertas.

El primer paso es decidir qué información se va a desechar de la celda de estado C_{t-1}. Esta decisión se determina mediante una capa sigmoidal llamada *forget gate layer* (en rojo en la gráfica). Los valores de entrada que toma son el estado oculto anterior y la entrada actual, es decir, h_{t-1} y x_t. Se obtiene un número entre 0 y 1, ya que se utiliza un función de activación sigmoidal, para cada número en la celda de estado C_{t-1}. Una salida de 1 representa rotundamente que hay que mantener dicha información y 0 que hay que eliminarla. La salida de esta capa se representa mediante el parámetro f_t y viene definido como

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

El siguiente paso es decidir qué nueva información se va a añadir a la celda de estado. Este paso consta de dos partes. Por un lado, una capa sigmoidal llamada *input gate layer* (en azul en la gráfica) que decide qué valores nuevos se van a añadir a la celda de estado. Por el otro lado, se encuentra una capa tangente hiperbólica que crea los nuevos valores de los posibles candidatos a ser añadidos a la celda de estado. Este resultado se denota como $\tilde{C}_t$. La información a incluir en la celda de estado es la combinación de las siguientes dos salidas

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

Después, hay que actualizar la antigua celda de estado C_{t-1} para obtener el nuevo estado C_t mediante la combinación de los dos pasos realizados. Se multiplica el antiguo estado C_{t-1} por f_t, olvidando las características innecesarias. A este resultado se le suma $i_t * \tilde{C}_t$, que son las propiedades que se quieren añadir. La nueva celda de estado resultante es

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

El último paso es devolver la salida de la celda LSTM. Se ejecuta una capa sigmoidal que decide qué información se va a devolver de la celda de estado. A esta expresión se le denomina *output gate* y viene especificada en amarillo en la gráfica. El estado actual pasa por una capa tangente hiperbólica y se multiplica por la salida obtenida en la capa sigmoidal. De esta manera solo se devuelve la información deseada.

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$
$$h_t = o_t \cdot \tanh(C_t)$$

Las redes LSTM se entrenan de la misma manera que las redes feed-forward, utilizan el algoritmo de *backpropagation* junto con el descenso del gradiente para la actualización de los pesos [23].

Capítulo 4

Desarrollo

Establecidas las bases teóricas, se procede a describir los elementos necesarios para la elaboración del asistente virtual. Para ello, se comienza explicando en qué consiste el software de IDbox y las funciones que se han decidido implementar para este primer asistente acorde a sus capacidades. Luego, se explica cómo se realiza la instalación de Rasa y los diferentes archivos que hay que completar tanto de estructura como de formatos para elaborar el corpus de entrenamiento. Finalmente se indica cómo se realiza el entrenamiento de los modelos.

4.1. IDbox

IDbox es un software basado en la inteligencia de negocios. Este término hace referencia a la utilización de diferentes procesos, aplicaciones y tecnologías que facilitan la obtención rápida y sencilla de los datos provenientes de sistemas o sensores para su posterior análisis. El software ofrece información en tiempo real para la supervisión y control de los distintos entornos y procesos de un negocio. Los negocios monitorizan los datos con diferentes sistemas, donde habitualmente cada equipo requiere de un software preciso. Los datos a su vez provienen de diferentes fuentes y formatos. Es por esto que tener una visión completa de lo que ocurre en una planta no es una labor sencilla.

IDbox permite recopilar los datos de diferentes orígenes, procesarlos de acuerdo a los parámetros previamente definidos y analizarlos de manera rápida, fácil e intuitiva.

En IDbox, las variables o de los datos recopilados recibe el nombre de *señal*.

4.2. Descripción del alcance del bot

Se ha tenido que crear una base de datos acorde a los objetivos establecidos. Para ello se tuvo que estudiar la sintaxis y los formatos que aceptaba la herramienta y diseñar a qué tipo de preguntas iba a tener que hacer frente el bot.

Aparte de las consultas particulares que se quieran abordar, las cuales se vienen definidas a continuación, el chatbot debe tener implementadas las funciones básicas de saludo, despida y tiene que devolver un mensaje por defecto en caso de no haber entendido la frase o pregunta de entrada.

Por lo tanto, el alcance del chatbot y las funcionalidades desarrolladas han sido las siguientes:

- El bot debe ser capaz de entender los saludos y responder con un saludo. Debe actuar de igual modo si el usuario se despide o le da las gracias.
- Debe ser capaz de entender si se le está preguntando por la representación de una gráfica, si quiere información sobre una señal específica o si lo que desea es un análisis descriptivo de esta. Se le puede indicar de manera opcional la franja temporal para los casos de la representación gráfica y el análisis descriptivo.
- Tiene que poder preguntar por la señal si el usuario no se la ha proporcionado. También debe preguntar qué tipo de gráfica desea si el usuario no la ha indicado.
- El bot debe ser capaz de manejar los errores de ortografía del usuario.
- El bot tiene que preguntar al usuario si la conversación ha ido bien *(feedback)*.
- En caso de no entender la petición del usuario, el bot tiene que sugerir la reformulación de la frase mostrando las posibles peticiones que se le pueden realizar (*fallback*).

Las dos bases de datos de las que extrae la información el asistente virtual corresponden a datos recopilados por una empresa anónima. Hay un total de 13 señales definidas desde *CIC0001* hasta *CIC0013*. Son variables de diferente carácter y su descripción, unidades y tipo vienen recogidos en el archivo `cic_descripcion.csv`. En la segunda base de datos, llamada `cic.csv`, se recogen las medidas de las señales que corresponden a la franja temporal entre noviembre del 2016 y noviembre del 2018 en intervalos de una hora.

4.2.1. Instalación

Rasa es una librería de código abierto desarrollada muy recientemente, en el año 2018. Al tratarse de una herramienta muy actual se encuentra en constante desarrollo, los commits en su repositorio son prácticamente diarios y la propia página web de Rasa cambia regularmente. Teniendo en cuenta todo lo anterior, es posible que el código adjuntado a lo largo del trabajo y en [24] haya quedado obsoleto y de fallos si se instala la versión más reciente de Rasa. Para el desarrollo del chatbot la versión utilizada de Rasa ha sido la 1.9.6 por lo que se recomienda instalar esta.

El lenguaje de programación que utiliza Rasa es Python, donde la versión utilizada ha sido la 3.7.7. Por otro lado, se necesita disponer además de la herramienta *Visual C++ Build Tools*.

La instalación de Rasa se realiza introduciendo la siguiente línea de comando en la terminal

```
pip install rasa==1.9.6
```

Para iniciar ahora un proyecto de Rasa se ejecuta este comando

```
rasa init --no-prompt
```

Se crean los archivos que se muestran en la Figura 4.1.

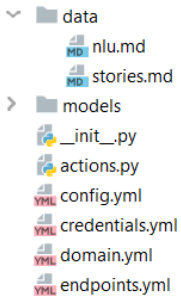

Figura 4.1: Archivos que se crean al iniciar un proyecto de Rasa.

Este comando crea todos los archivos necesarios para iniciar un proyecto de Rasa y entrena un simple bot con algunos datos de muestra por defecto. El modelo entrenado se guarda en la carpeta `models`.

Los archivos que conforman el corpus de entrenamiento son tres: `nlu.md`, `stories.md` y `domain.yml`. El formato más utilizado para estos archivos, salvo en el caso del dominio, es *Markdown* ya que normalmente es más sencillo trabajar con él, aunque también se pueden escribir en *JSON*.

Aparte del archivo del dominio y el de configuración, este útlimo ya descrito en el capitulo anterior, hay otros dos archivos que se utilizan para conectar el bot a diferentes canales. En este caso, todos estos archivos están escritos en el formato YAML (*Yet Another Markup Language*).

En el archivo de Python llamado `actions` se implementan las acciones personalizadas.

A continuación, se definen y explican los diferentes archivos que conformaran el repositorio. Los archivos de referencia que se muestran no están completos, sirven para visualizar la estructura y formato que deben tener. Tanto estos archivos como el código al completo implementado se puede consultar en el siguiente repositorio [24].

4.2.2. Frases de entrenamiento: `nlu.md`

Este archivo está formado por los posibles mensajes que puede escribir el usuario. En él hay que indicar la correspondiente intención y las entidades que se encuentran en la oración. A partir de estas frases de referencia el bot aprende a identificar las intenciones y entidades. Además de esto, hay dos apartados adicionales que son el de sinónimos y expresiones regulares.

El primero de ellos sirve para asociar diferentes entidades a un mismo valor. Es lo que utiliza la componente `EntitySynonymMapper` del pipeline ya descrita.

El segundo mencionado, las expresiones regulares, se puede utilizar para facilitar la extracción de entidades. Por ejemplo, si la entidad tiene una estructura predeterminada (como puede ser el caso de la señal, cuatro letras seguido de cuatro dígitos), se utiliza una expresión regular para facilitar la detección de dicha entidad. Estas expresiones se utilizan en la componente `RegexFeaturizer`.

El formato que debe tener este archivo es el siguiente:

- Se debe indicar el nombre de la intención.
- Las entidades deben ir entre corchetes y entre paréntesis se debe indicar el nombre de la entidad a la que pertenecen.
- Se debe indicar a qué tipo de entidad pertenecen tanto los sinónimos como las expresiones regulares.

```
## intent:informacion
- ¿Puedes proporcionarme la información de la señal [cic0001](senal)?
- Quiero saber de que [tipo](tipo) es la señal [cic0007](senal)
(fecha) y [2018-05-05](fecha)
- Dame la [unidad](unidad) y la [descripcion](descripcion) de la señal
[CIC0008](senal)
...

## synonym:unidad       ## regex: senal
- unidades             - [a-zA-Z]{3}[0-9]{4}
```

4.2.3. Dominio: `domain.yml`

El dominio es el entorno donde el asistente opera. En él se incluyen las respuestas del bot, el nombre de las acciones personalizadas creadas, las intenciones, entidades y slots que el bot debe conocer.

Los conceptos nuevos aquí son las acciones personalizas y los slots. Las acciones personalizadas son funciones que pueden ejecutar cualquier código que se quiera, de este modo, la respuesta no tiene por qué ser fija y puede ser acorde al mensaje del usuario. Los *slots* son la memoria del bot. Actúan como un almacén de clave-valor que se utiliza para almacenar información (entidades) que el usuario ha facilitado, como el nombre de una señal. Por lo general, se desea que estos slots influyan en el diálogo, por lo que son muy utilizados en las acciones personalizadas. Los diferentes tipos de slots son los siguientes: texto, booleano, categórico, flotante, lista y sin característica (*unfeaturized*).

Por último, en este archivo se puede definir el periodo de inactividad a partir del cual se inicia una nueva sesión de conversación. Mediante el parámetro `session_expiration_time` se define el tiempo de inactividad en minutos. Otro de los parámetros que se puede declarar en este archivo es `carry_over_slots_to_new_session`. Mediante un valor booleano se determina si se quiere que los slots se olviden entre sesiones diferentes.

Es este archivo, tal y como se puede observar, hay que que recapitular en forma de lista todas las intenciones, acciones, entidades, slots y respuestas que se hayan definido.

```
session_config:
  session_expiration_time: 10
  carry_over_slots_to_new_session: true

intents:            actions:                    entities:
- saludo            - utter_saludo              - senal
- despedida         - utter_despedida           - tipo_grafica
- grafica           - action_grafica_hist       - fecha
- ...               - ...                       - ...

slots:                        responses:
  senal:                        utter_saludo:
    type: text                  - text: Hola Hola
  tipo_grafica:                 - text: Buenos días, ¿qué necesitas?
    type: categorical          utter_despedida:
    values:                     - text: ¡Adiós!
    - serie                     - text: ¡Hasta pronto!
    - histograma                - text: ¡Ciao, ciao!
  ...                           ...
```

4.2.4. Historias: `stories.md`

Las historias son simulaciones de una conversación real entre el bot y el usuario. Es la estructura de los datos que se utiliza para entrenar el modelo de gestión del diálogo. Los mensajes del usuario se convierten en las intenciones correspondientes mientras que las respuestas se expresan con el debido nombre de la acción. Se muestra a continuación un ejemplo de una conversación real con el asistente y el formato equivalente que utiliza el bot para entrenarse.

Usuario: ¡Buenos días!	`* saludo`
Bot: Hola	`- utter_saludo`
U: Dame la gráfica del id cic0005	`* grafica`
B: ¿Qué tipo de gráfica desea?	`- utter_ask_tipo_grafica`
U: Un histograma	`* grafica_hist`
B: Aquí tiene su gráfica: (imagen)	`- action_grafica_hist`
U: Muchas gracias	`* agradecer`
B: De nada	`- utter_agradecer`
U: Hasta pronto	`* despedida`
B: Adiós	`- utter_despedida`

4.2.5. Acciones personalizadas: `actions.py`

Como ya se ha mencionado anteriormente, si por ejemplo el usuario saluda, el bot tiene que responder con un texto predefino de saludo que se encuentra especificado en la archivo del dominio. Sin embargo, no siempre la respuesta es fija, a veces se tiene que utilizar la información que proporciona el usuario para devolver una respuesta acorde a los datos facilitados.

Las acciones personalizadas son respuestas del asistente que incluyen algún código personalizado. Este código personalizado puede ser desde una simple respuesta de texto hasta, como ocurre para el chatbot desarrollado, conectarse a una base de datos externa, extraer los datos deseados y devolver una gráfica.

Las funciones personalizadas que se han creado para el bot de IDbox han sido cuatro: retornar detalles de una señal, imprimir un análisis descriptivo de los datos, devolver una gráfica y en caso de haber un problema de comprensión lanzar una frase por defecto.

Las dos primeras se han implementado utilizando la librería de *Pandas* para el manejo de dataframes.

La implementación de la tercera, la impresión de las gráficas, ha sido algo más compleja ya que Rasa no permite devolver imágenes locales, por lo que ha sido necesario implementar una API que cree una URL para cada solicitud del usuario. Para ello, primero se ha creado una carpeta llamada `img`, donde se guardan las gráficas generadas utilizando la librería *Matplotlib*

referentes a la solicitud del usuario. La API en cuestión, crea un servidor local asociado a la imagen que se quiere imprimir, pudiendo utilizar la URL generada para la impresión de la gráfica.

Para finalizar, la componente `FallbackPolicy` del apartado del manejo del diálogo se encarga de comprobar si los umbrales establecidos tanto para el entendimiento como la gestión del diálogo se sobrepasan por la predicción o no y en caso de que no imprimir la frase por defecto establecida. Para definir esta función, simplemente hay que escribir cuál es la frase que se desea devolver por defecto.

4.2.6. Archivos `credentials.yml` y `endpoints.yml`

El archivo de credenciales sirve para que el asistente esté disponible en plataformas de mensajería como pueden ser Facebook Messenger, Telegram, etc. Para este trabajo no se ha realizado esta labor pero apuntar que Rasa permite conectar el bot a diferentes servicios. Es por esto que dicho archivo no debe ser modificado y debe quedarse como se encuentra por defecto.

Un endpoint es un extremo final de un canal de comunicación. Cuando una API interactúa con otro sistema, bien para acceder a los recursos o enviar solicitudes, los puntos de contacto de esta comunicación se consideran endpoints. En el caso de las APIs, un endpoint puede incluir una URL de un servidor. Para este caso lo único que se ha utilizado y necesita ser descomentado es el endpoint de las acciones para iniciar el servidor de las acciones personalizadas.

4.2.7. Entrenamiento

Ejecutando el siguiente fragmento de código en la terminal se entrenan los diferentes modelos pertenecientes a Rasa NLU y Core y los almacena en el directorio llamado `models`.

```
rasa train
```

Uno de los argumentos opcionales disponibles es `--augmentation` que sirve para crear historias más largas juntando al azar historias del archivo de entrenamiento. Esto suele mejorar el rendimiento ya que se quiere enseñar al bot a ignorar la cronología del diálogo cuando esta no es relevante y responder con la misma acción correspondiente sin importar lo que haya sucedido antes.

El aumento de datos puede ser modificado estableciendo el indicador de aumento deseado. Poner este parámetro en 20 significa que se crean 200 historias adicionales. Hay que tener en cuenta que su uso aumenta el número de historias a procesar, por lo que también el tiempo de ejecución.

Capítulo 5

Análisis de los modelos y resultados

En este capítulo se detallan las estructuras de los modelos que se han utilizado particularmente para el desarrollo del bot así como los respectivos resultados de precisión. Se comienza describiendo la razón de porqué se ha elegido un pipeline supervisado.

5.1. Elección del pipeline

Recordemos que existen dos tipos de estructuras para el apartado del procesamiento del texto que son las configuraciones en las que se utilizan modelos pre-entrenados y en las que no. Las razones de por qué se ha elegido una estructura supervisada, es decir, en la cual no se utilizan incrustaciones pre-entrenadas, son fundamentalmente dos.

En primer lugar, el bot que se desea desarrollar se trata de un simple prototipo por lo que de momento no se ha visto oportuno introducir modelos pre-entrenados. De esta manera además, se agiliza el entrenamiento.

En segundo lugar, como se ha mencionado en el Capitulo 3, estos modelos pre-entrenados están desarrollados principalmente para el inglés, por lo que para el resto de idiomas puede que sea escaso su rendimiento. Para respaldar esto, cuando se empezó a trabajar en el desarrollo del bot se utilizó la versión de Rasa NLU 0.12.3, referente a la versión de Rasa 1.7.0. Con esta versión, se pudo observar cómo se obtenían niveles de confianza mucho más inferiores utilizando un pipeline pre-entrenado que supervisado. Mencionar que en esta versión todavía no se había implementado el modelo DIET y que para la extracción de las entidades se utilizaban campos aleatorios condicionales, pero en cambio para la clasificación de la intención se empleaban máquinas de vectores de soporte. Actualmente, la versión de Rasa NLU que se está utilizando es la 0.15.1 y los niveles de confianza han aumentado considerablemente gracias al modelo de DIET. A continuación

se muestran para algunas oraciones de entrada los niveles de confianza tanto para una estructura pre-entrenada como supervisada.

Entrada de texto	Intención	Pre-entrado	Supervisado
hollaa	`saludo`	0.93	0.99
Muchisisimas gracias!!	`agradecer`	0.79	0.85
qué bien!	`buen_humor`	0.98	0.99
Hasta la proxima vez	`despedida`	0.55	0.96
Necesito ahora 1 gráfca	`grafica`	0.86	0.90
dame info d una señl	`informacion`	0.99	0.93
Imprímeme la mdia de un id	`descripcion`	0.06	0.84

Tabla 5.1: Nivel de confianza de la intención para diferentes entradas de texto para un pipeline pre-entrenado y supervisado.

Se observa como en la mayoría de los casos se obtienen niveles de confianza similares aunque para un par de entradas la diferencia es notoria y la estructura supervisada lo ejecuta mejor. Es por esto, visto que el bot a implementar todavía es básico y que la configuración supervisada se ejecuta correctamente, incluso mejor en algunas situaciones, no se ha visto la necesidad de introducir componentes pre-entrenadas.

5.2. Análisis detallado de los modelos

Rasa proporciona una estructura predetermina de los modelos que puede ser customizada por el usuario cambiando los parámetros que vienen por defecto. Se probó a entrenar el bot con dichos parámetros y tal y como se mostrará en la sección 5.3 los valores de precisión obtenidos han sido buenos y el tiempo de ejecución no ha sido alto. Es por esto, y debido a los pocos datos de entrenamiento que se tenían disponibles, solo se ha visto necesario cambiar el número de épocas de entrenamiento para la red LSTM reduciéndolo a 100 en vez de 300.

A la hora de ejecutar el comando de entrenamiento el primer modelo que se entrena es el del manejo del diálogo, que como ya se sabe corresponde a la red LSTM. La estructura del modelo, tal y como se puede observar en la Figura 5.1, está compuesta de cuatro capas diferentes:

1. Una máscara que se encarga de hacer que todas las entradas sean de igual longitud que la entrada de dimensión mayor. Se observa que sus parámetros son $(5, 65)$. El primer número hace referencia al número de historias pasadas que mira (`max_history`), que por defecto es 5. Si se quiere variar dicho número, al igual que en la componente `MemoizationPolicy` hay que introducir el parámetro `max_history` y asignarle el valor deseado. La cifra de 65 hace referencia al número de

acciones (16 propias y 9 por defecto), entidades (9), intenciones (13) y slots (18[1]) total.

2. Una capa LSTM compuesta de 32 celdas.

3. Una capa densa compuesta por 25 neuronas (acciones posibles).

4. Una función de activación softmax, que asigna una probabilidad a cada acción de tal manera que todas ellas sumen 1. De esta forma, la acción con mayor probabilidad será la resultante.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
masking (Masking)            (None, 5, 65)             0
_________________________________________________________________
lstm (LSTM)                  (None, 32)                12544
_________________________________________________________________
dense (Dense)                (None, 25)                825
_________________________________________________________________
activation (Activation)      (None, 25)                0
=================================================================
Total params: 13,369
Trainable params: 13,369
Non-trainable params: 0
```

Figura 5.1: Estructura de la red LSTM.

Además el modelo se entrena utilizando la función de pérdida categórica de entropía cruzada, para la actualización de los pesos utiliza el algoritmo de descenso del gradiente con momento *RMSprop* con un ratio de aprendizaje del 0.001 y se entrena para 100 épocas.

Para el caso del modelo de DIET, tal y como se muestra en el Capítulo 3, el pipeline establecido no se utiliza para entrenar un modelo de lenguaje. De momento se tiene poco corpus de entrenamiento y por ahora tampoco es el objetivo entrenar un modelo de lenguaje, por lo que esta funcionalidad se ha desactivado. Además, ningún extractor de características pertenece a un modelo pre-entrenado por lo tanto la parte en la que los tokens atraviesan una red pre-entrenada se obvia también. En consecuencia, el modelo final que se entrena para el pipeline elegido se observa en la Figura 5.2.

Por ejemplo, la estructura de la red transformadora está compuesta de dos capas y devuelve como salida un vector de dimensión 256. Igualmente, las capas de incrustación están compuestas de una red densa de 20 neuronas. Para más información del resto de configuraciones que conforman la estructura de DIET se puede consultar [25].

[1]Para los slots de tipo categórico, hay un valor por defecto denotado como `__other__`. Todos los valores encontrados que no están explícitamente definidos en el dominio para el slot categórico dado se asignan a `__other__` para su categorización.

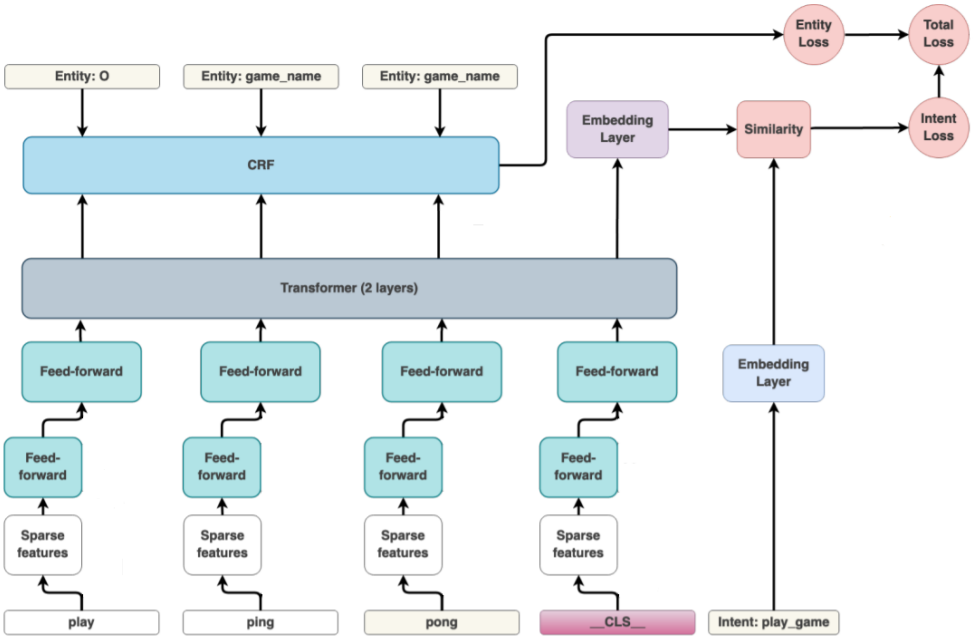

Figura 5.2: Modelo DIET empleado en el desarrollo del bot.

En este caso se utiliza el optimizador de Adam, para una tamaño de batch que se incrementa linealmente para cada época desde 64 hasta 256, con un ratio de aprendizaje del 0.001 y para 100 épocas también.

5.3. Resultados

Para el entrenamiento de los modelos se han utilizado los ficheros que se encuentran en la carpeta `data`. Para la validación de ambos modelos se han escrito frases que no se encuentran en el corpus de entrenamiento para todas las intenciones y se han creado unas conversaciones. Ambos ficheros de validación se encuentran en este caso en la carpeta llamada `results`. En el fichero de validación para el modelo de DIET se han introducido oraciones con faltas de ortografía, erratas y abreviaturas para comprobar si pequeñas variaciones en el conjunto de entrada afecta a su ejecución.

Aunque esta no es la configuración habitual para la división de los datos en el campo de la inteligencia artificial, donde los conjuntos de entrenamiento y validación son estadísticamente equivalentes, para el caso del desarrollo de chatbots es necesario hacerlo así. No es posible realizar una partición aleatoria del conjunto de entrenamiento para la validación ya que oraciones tan habituales y fundamentales para el aprendizaje del modelo como son *Hola*, *Gracias* o *Adiós* pueden quedarse fuera de él y necesitan incluirse obligatoriamente. Es por esto que la creación de las oraciones de validación

se ve condicionada a las introducidas para el entrenamiento. Se exponen primero los resultados obtenidos para el modelo DIET y luego para el modelo de la LSTM.

Para evaluar el modelo para la clasificación de la intención y extracción de entidades tanto en en el conjunto de entrenamiento como en el de validación se debe introducir la siguiente linea de comando

```
rasa test nlu  --nlu NLU --model MODEL --successes
```

donde en `NLU` se debe introducir la ruta del fichero, bien de entrenamiento o de validación, y en `MODEL` la ruta del fichero del modelo. Mediante este comando se crean varios archivos cuyos resultados se analizan a continuación.

En el Apéndice A, la primera matriz que se muestra en las secciones de Entrenamiento A.1 y Validación A.2 es la matriz de confusión de las intenciones (Figuras A.1 y A.3). Se puede ver como los datos de entrenamiento los aprende correctamente y no comete ningún fallo en la clasificación. Por el contrario en la validación, tal y como se puede observar también en el fichero `intent_errors.json` que se encuentra en la ruta `results/test/nlu`, los pocos errores que se encuentran provienen en su mayoría de las intenciones elementales como `buen_humor` y `agradecer`. Esto ha ocurrido debido a que a la hora de crear los datos de validación se han tenido que buscar frases menos comunes. Aparte de ser pocas las opciones para este tipo de intenciones respecto a las formas de escribir una solicitud, se han recogido todas ellas en los datos de entrenamiento. Además, no se espera que el usuario vaya a introducir algo muy diferente a lo que se encuentra en el entrenamiento y que se cometan entonces tantos errores de clasificación.

En la Tabla 5.2 se recoge la confianza media obtenida en todas las frases para sus respectivas intenciones tanto para los datos de entrenamiento como de validación.

Intención	**Confianza media en train**	**Confianza media en test**
`saludo`	0.99	0.77
`buen_humor`	0.99	0.61
`agradecer`	0.99	0.45
`despedida`	0.99	0.91
`grafica`	0.98	0.77
`grafica_serie`	0.98	0.89
`grafica_hist`	0.99	0.76
`informacion`	0.99	0.84
`descripcion`	0.99	0.86
`dar_senal`	0.99	0.82

Tabla 5.2: Confianza media para cada intención.

Se observa como las intenciones que peores resultados obtienen en la validación son las básicas y en cambio las funcionalidades implementadas para este bot como son `grafica_serie` o `descripcion` las identifica con una confianza media de más del 0.75. Esto se trata de un resultado muy positivo ya que por ejemplo como se ha mencionado, se tienen diferentes maneras de escribir la solicitud de una gráfica, y las ha detectado correctamente. Sin embargo, para el resto de intenciones se han cometido errores por utilizar oraciones poco frecuentes en el lenguaje escrito. Hay que tener en cuenta que como se disponen de pocos datos de entrenamiento, en consecuencia también de validación, los errores afectan notoriamente. Además, las oraciones de validación representan un caso más pesimista y alejado de la realidad, ya que se han tenido que introducir frases más atípicas de las que se introducirían.

Se han escrito algunas peticiones fuera del alcance del bot para observar cómo el nivel de confianza de la intención con la que la clasifica la oración reduce considerablemente. Como se puede observar en la Tabla 5.3 ninguna de las peticiones supera el 0.42 de confianza, por lo que se devolvería para todos estos casos la acción *fallback*. Con esto se quiere mostrar que cuando se le realizan peticiones fuera de su alcance, aún introduciendo entidades con las que ha sido entrenado, identifica que son funcionalidades que no tiene implementadas ya que los niveles de confianza bajan notablemente.

Entrada de texto	**Confianza**
¿Cuál es la ubicación de la empresa?	0.41
Créame una notificación para la señal cic0010	0.40
¿Qué horario tenéis?	0.27
¿Qué valor marca actualmente el tag cic0009?	0.27
Déjame un email de contacto	0.39
Crea el documento correspondiente al id CIC0009	0.26
¿Cuál es tu nombre?	0.31
Facilítame el nº de contacto de área de monitorización	0.38

Tabla 5.3: Confianza para entradas de texto fuera del alcance del bot.

Respecto a la extracción de las entidades se tiene que no se ha cometido ningún fallo en los datos de entrenamiento ya que no se ha creado en su evaluación el archivo `DIETClassifier_errors.json`. Los dos únicos errores que se han cometido en la validación han sido por escribir únicamente los números para una señal y la palabra *sre* en vez de *serie*. Aunque en las frases de validación se hayan introducido erratas, se espera que el usuario no las introduzca ya que de lo contrario tampoco se ejecutaría bien la acción. Por ejemplo, el bot no captaría de qué señal quiere la gráfica si esta está mal escrita. Por lo tanto se puede concluir que las entidades se extraen satisfactoriamente.

Tanto la correcta identificación de las intenciones como la extracción de

las entidades llevada a cabo por el modelo DIET ocurre principalmente por las dos siguientes razones:

- Por el momento el alcance del bot es limitado, no tiene muchas funciones implementadas y estas se pueden diferenciar fácilmente ya que se necesitan utilizar palabras claves para su solicitud, como pueden ser: gráfica, información, análisis estadístico, etc. Si el modelo consigue identificar estas palabras, cosa que hace, la intención es clara. Además, para el caso de identificar las intenciones como pueden ser el saludo o la despedida, las opciones de expresión son muy restringidas, por lo que las más comunes vienen indicadas en el entrenamiento para su correcta clasificación.

- Se han introducido expresiones regulares para la extracción de las entidades, por lo que se facilita la identificación de estas potenciando la correcta ejecución del modelo.

Para evaluar el modelo del manejo del diálogo hay que introducir el siguiente comando indicando de nuevo la ruta del fichero de los datos a evaluar y el modelo deseado.

```
rasa test core --stories STORIES --model MODEL
```

Tanto para los datos de entrenamiento como de validación se observa en las Figuras A.2 y A.4 como se obtiene una matriz diagonal con todas las acciones correctamente clasificadas.

La principal razón de por qué no cometen fallos este modelo es porque dada una intención concreta con unos slots dados se quiere que se responda siempre con la misma acción. Por ejemplo, si el usuario saluda, el bot debe responder saludando, si no se proporciona para una petición específica la señal, esta debe ser solicitada primero antes de pasar a la siguiente acción, etc. En el momento que el modelo de la LSTM "entiende" estos patrones siempre se tiene que responder con la misma acción.

Finalmente, para probar el bot y establecer una conversación con él se puede hacer o bien desde la terminal o utilizando la interfaz que proporciona Rasa X. En el Apéndice B se puede observar una conversación completa de prueba.

Se aprecia como los resultados de dicha conversación son satisfactorios y responde correctamente a todas las peticiones formuladas. Además, se le pregunta por la temperatura en Santander (función para la que no ha sido programado) y solventa adecuadamente la petición devolviendo una frase por defecto para solucionar adecuadamente la situación.

Capítulo 6

Conclusiones y trabajo futuro

El objetivo principal de esta memoria ha sido desarrollar un chatbot para la empresa en la que he realizado las prácticas que fuera capaz de resolver ciertas peticiones. Todas las funcionalidades propuestas se ha visto que se ejecutan satisfactoriamente con un prototipo totalmente operativo y funcional.

Además, se quería ver que la tecnología de código abierto de Rasa era lo suficientemente competente para la realización del bot, en comparación con herramientas muy potentes que hay en el mercado actualmente como puede ser Dialogflow de Google. A la empresa le interesa que poco a poco se vayan añadiendo progresivamente más funcionalidades y que los usuarios puedan hacer todas las peticiones que deseen sin costes añadidos una vez que el bot está implementado. No hay más que ver las matrices de confusión para observar que el framework de inteligencia artificial de Rasa ha cumplido con todas las expectativas.

Como trabajo futuro se podría implementar progresivamente más funciones. Como ya se ha comentado, este ha sido un primer prototipo para afirmar que la tecnología de Rasa es una buena elección para implementar un bot. Asimismo, se ha mencionado en diversas ocasiones que Rasa se trata de una tecnología que se encuentra en constante actualización, lanzando nuevas mejoras e incluso nuevos modelos implementados por ellos mismos. Por ejemplo, en la actual versión de Rasa, en concreto la componente `KerasPolicy` que implementa una LSTM para el manejo del diálogo ha quedado obsoleta y se ha sustituido por una nueva componente basada en transformadores.

Sería muy favorable poder ejecutar el bot como una demo a nivel interno en la empresa para que el personal lo pudiese utilizar y comprobar su funcionamiento. Al fin y al cabo, todo el entrenamiento tiene que estar escrito por el propio desarrollador y puede estar condicionado a su manera de escribir o a la forma en la que él preguntaría al bot. De esta manera, podría verse diferentes puntos de vista para realizar las peticiones y si fuera necesario, añadir oraciones al entrenamiento que no se habían tenido en cuenta previamente

haciendo así el corpus más imparcial y general.

Finalmente, podría estudiarse la integración de tecnologías para que se pudiera utilizar como asistente de voz. Rasa ofrece documentación para utilizar las herramientas Mozilla DeepSpeech y Mozilla TTS para integrar el reconocimiento de voz. La primera de ellas, se encarga de transformar la entrada de audio del usuario a texto, la segundo en cambio, realiza la tarea contraria, toma el texto de la respuesta y lo convierte en audio. Ambas herramientas están escritas en Python, lo que facilita la integración con Rasa. Además, al igual que Rasa, se tratan de herramientas de código abierto y vienen con un conjunto de modelos pre-entrenados, aunque también se pueden entrenar modelos propios.

Apéndice A

Matrices de confusión

A.1. Entrenamiento

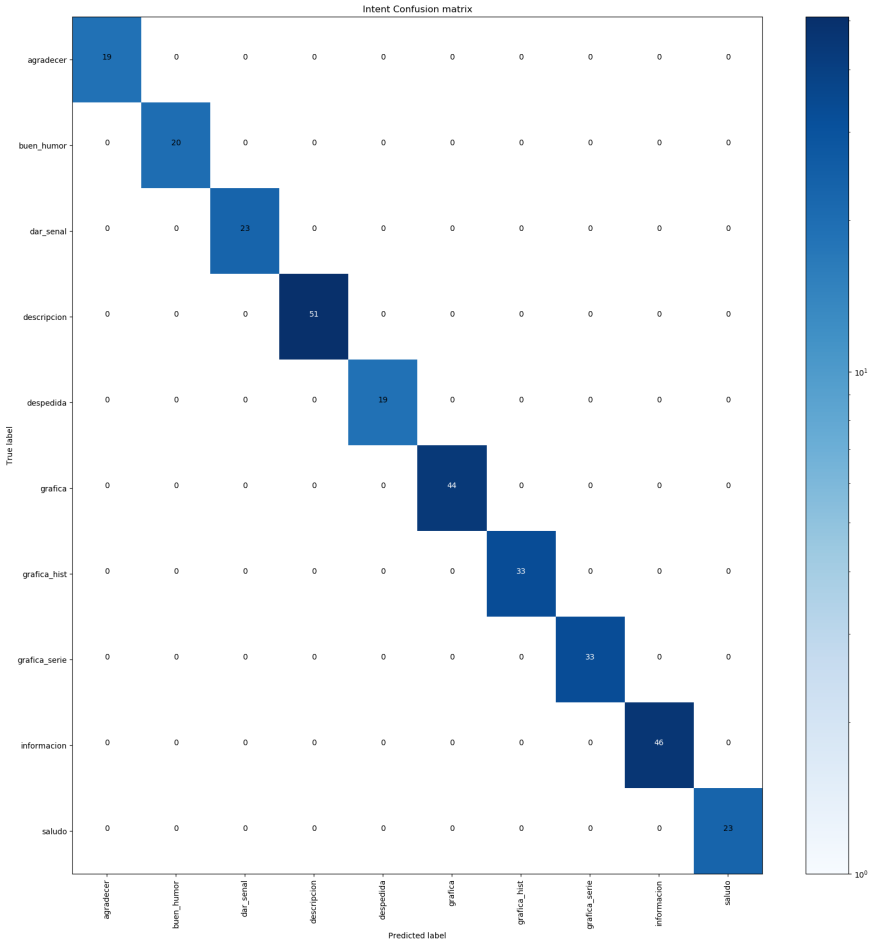

Figura A.1: Matriz de confusión de las intenciones.

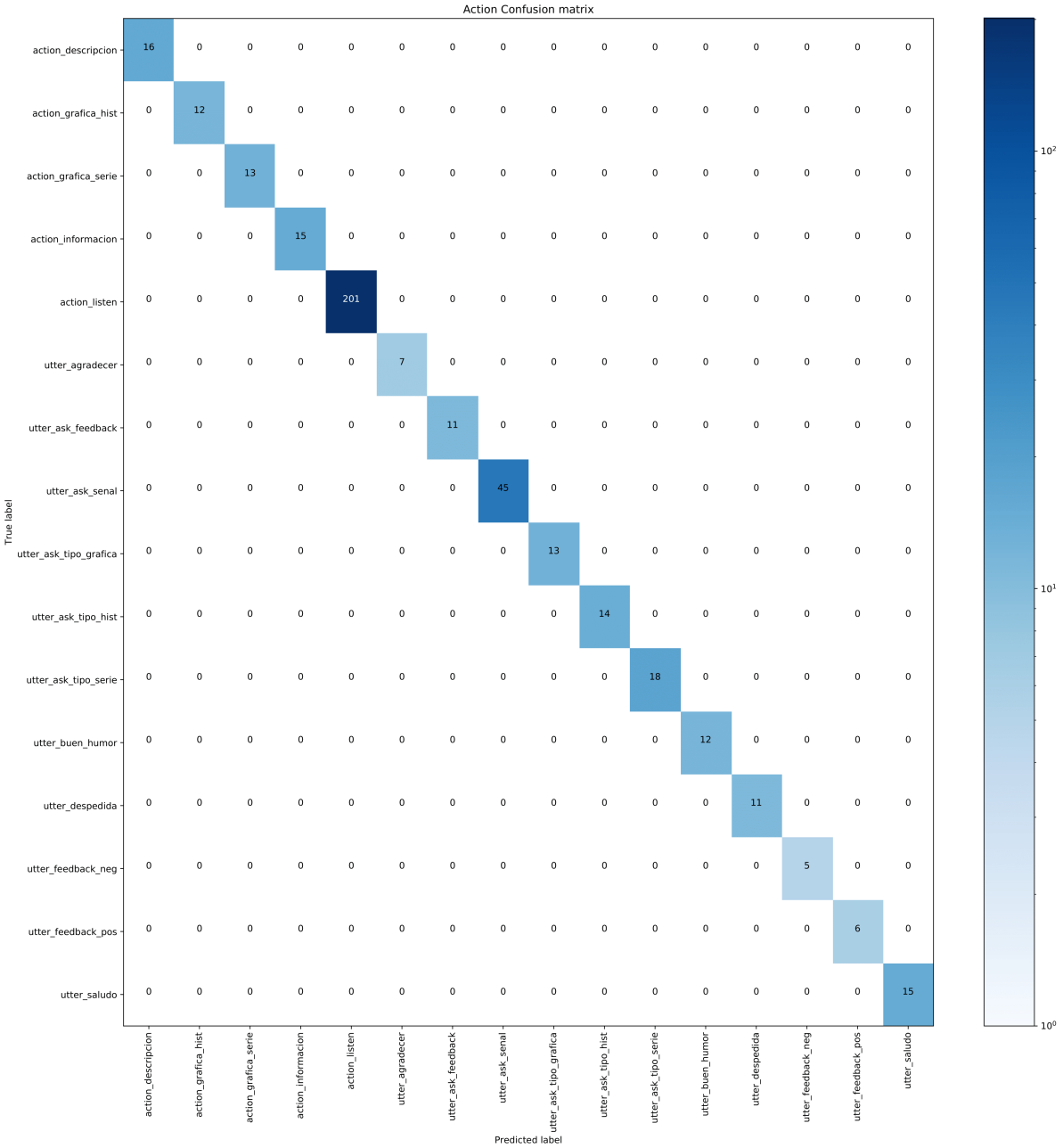

Figura A.2: Matriz de confusión de las acciones.

A.2. Validación

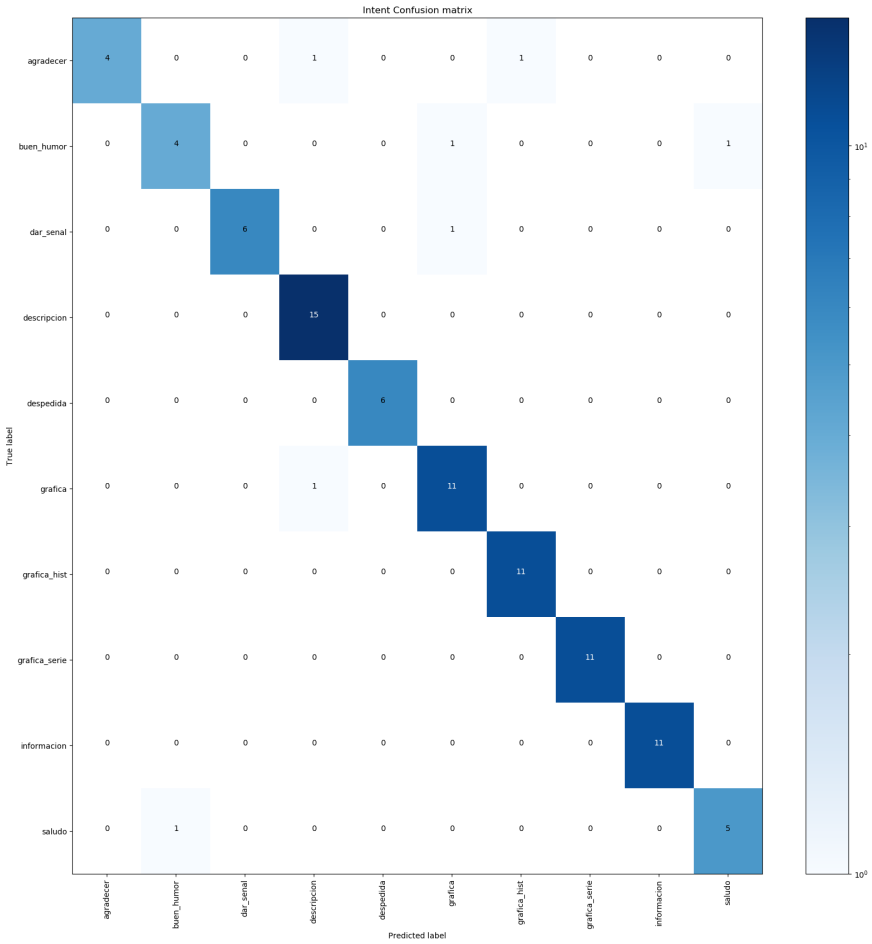

Figura A.3: Matriz de confusión de las intenciones.

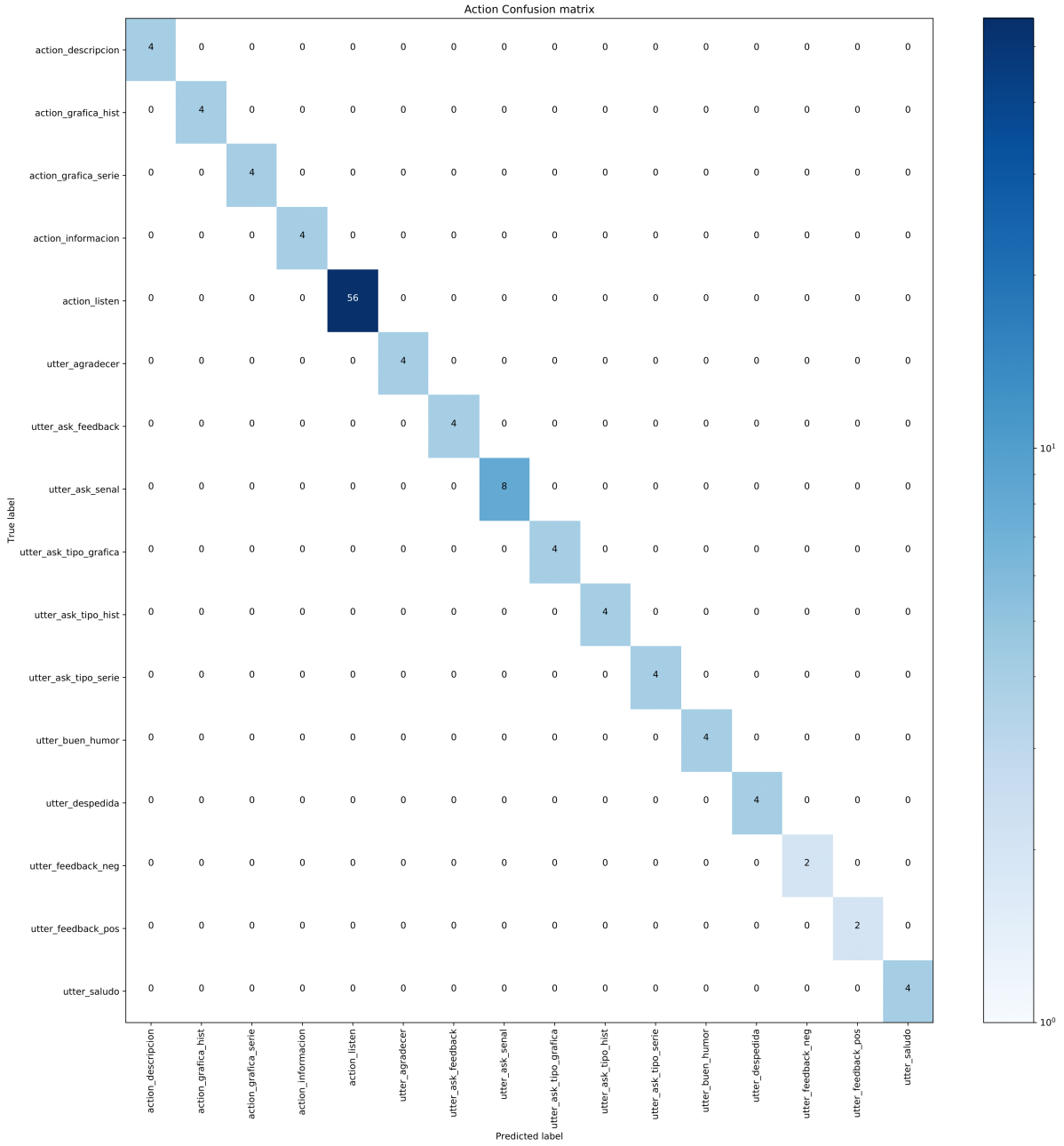

Figura A.4: Matriz de confusión de las acciones.

Apéndice B

Conversación con el bot

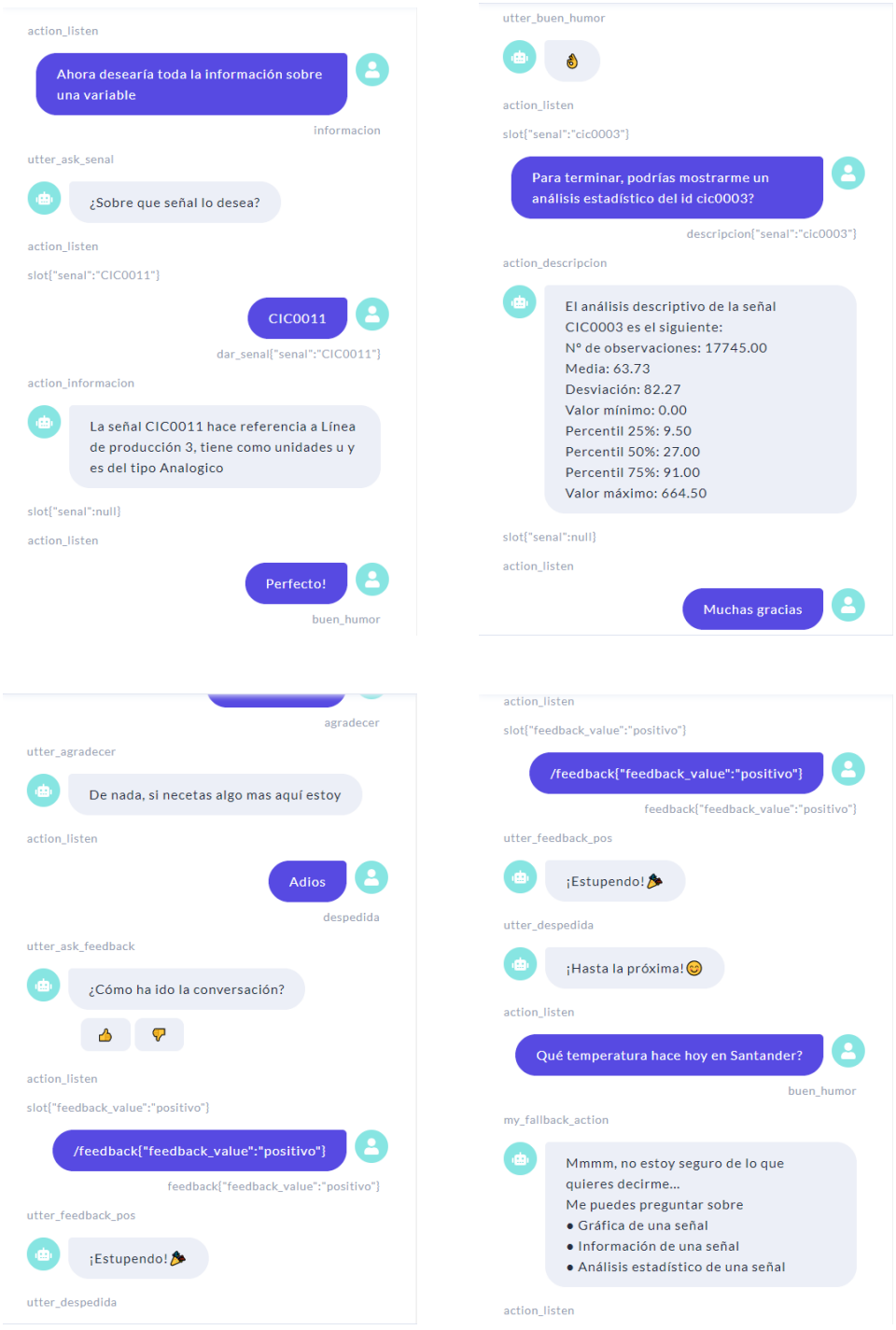

Figura B.1: Conversación completa con el bot.

Bibliografía

[1] Zumstein, Darius and Hundertmark, Sophie. CHATBOTS–AN INTERACTIVE TECHNOLOGY FOR PERSONALIZED COMMUNICATION, TRANSACTIONS AND SERVICES. *IADIS International Journal on WWW/Internet*, volume 15(1), 2017.

[2] Brandtzaeg, Petter Bae and Følstad, Asbjørn. Why people use chatbots. *International Conference on Internet Science*, pages 377-392. Springer, 2017.

[3] Rajai Nuseibeh. What is a Chatbot?. *Chatbots Magazine*, 2018. [online] Available at: `https://chatbotsmagazine.com/what-is-a-chatbot-6dfff005bb34`

[4] Sansonnet, Jean-Paul and Leray, David and Martin, Jean-Claude. Architecture of a framework for generic assisting conversational agents. *International Workshop on Intelligent Virtual Agents*, pages 145-156. Springer, 2006.

[5] Raj, Sumit. *Building Chatbots with Python: Using Natural Language Processing and Machine Learning*. Apress, 2018.

[6] Turing, Alan M. Computing machinery and intelligence. *Parsing the Turing Test*, pages 23-65. Springer, 2009.

[7] Rasa Documentation. [online] Available at: `https://rasa.com/docs/`

[8] Rasa Open Source. GitHub repository. [online] Available at: `https://github.com/RasaHQ/rasa`

[9] Khurana, Diksha and Koli, Aditya and Khatter, Kiran and Singh, Sukhdev. Natural language processing: State of the art, current trends and challenges. *arXiv preprint arXiv:1708.05148*, 2017.

[10] Manning, Christopher D and Manning, Christopher D and Schütze, Hinrich. *Foundations of statistical natural language processing*. MIT press, 1999.

[11] Mikolov, Tomáš and Yih, Wen-tau and Zweig, Geoffrey. Linguistic regularities in continuous space word representations. *Proceedings of the 2013 conference of the north american chapter of the association for computational linguistics: Human language technologies*, pages 746-751, 2013.

[12] scikit-learn: Feature extraction. [online] Available at: `https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html`

[13] Mantha, Mady. Introducing DIET: state-of-the-art architecture that outperforms fine-tuning BERT and is 6X faster to train. *Rasa Research website*.

[14] Bunk, Tanja and Varshneya, Daksh and Vlasov, Vladimir and Nichol, Alan. DIET: Lightweight Language Understanding for Dialogue Systems. *arXiv preprint arXiv:2004.09936*, 2020.

[15] Sutton, Charles and McCallum, Andrew and others. An introduction to conditional random fields. *Foundations and Trends® in Machine Learning*, volume 4(4), pages 267-373. Now Publishers, Inc., 2012.

[16] Olah, Christopher. *Understanding lstm networks*, 2015. [online] Available at: `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`

[17] Devlin, Jacob and Chang, Ming-Wei and Lee, Kenton and Toutanova, Kristina. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[18] Pennington, Jeffrey and Socher, Richard and Manning, Christopher D. Glove: Global vectors for word representation. *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532-1543, 2014.

[19] Henderson, Matthew and Casanueva, Iñigo and Mrkšić, Nikola and Su, Pei-Hao and Vulić, Ivan and others. ConveRT: Efficient and Accurate Conversational Representations from Transformers. *arXiv preprint arXiv:1911.03688*, 2019.

[20] Vaswani, Ashish and Shazeer, Noam and Parmar, Niki and Uszkoreit, Jakob and Jones, Llion and Gomez, Aidan N and Kaiser, Łukasz and Polosukhin, Illia. Attention is all you need. *Advances in neural information processing systems*, pages 5998-6008, 2017.

[21] Murugan, Pushparaja. Learning the sequential temporal information with recurrent neural networks. *arXiv preprint arXiv:1807.02857*, 2018.

[22] Hochreiter, Sepp and Schmidhuber, Jürgen. Long short-term memory. *Neural computation*, volume 9(8), pages 1735-1780. MIT Press, 1997.

[23] Chen, Gang.A gentle tutorial of recurrent neural network with error backpropagation. *arXiv preprint arXiv:1610.02583*, 2016.

[24] Código desarrollado para la prueba de concepto del chatbot para IDbox. GitHub repository. [online] Available at: `https://github.com/vpinilla001/TFM`

[25] Rasa Documentation. NLU Componentes [online] Available at: `https://rasa.com/docs/rasa/nlu/components/#diet-classifier`